UNIVERSITY OF OSLO
Department of Informatics

Analyzing Sensor Data for Active Music

Masters thesis
(60 pt)

Roger Stein Grading

May 2, 2011

Abstract

This thesis is about analysis of motions for *active music* applications, where motions control music in real–time.
Motion data is derived from accelerations measured in (Euclidean) 3D by one accelerometer. In order to capture motions on different time–scales, a necessary preprocessing step for analysis is calibration and segmentation on the sensor data streams.
For sensor data analysis, a real–time, configurable motion classifier has been implemented. Datasets for the experiments with this classifier are based on two categories of equally sized pre–captured accelerations. Classification performance has been evaluated on a range of segment lengths (i.e. time–scales of motions)—each length corresponding to a unique dataset.
Regarding postprocessing of the classifications for sound control, two quite different mapping systems have been developed—to different extents. Both control different musical aspects, although at different intervals. The first system is trigger–based and inspired by the concept of *hypermusic* Machover [2004]. However, for reasons that will become apparent, further development of this system has been put on hold. The second (and latest) system is for multi–channel continuous normalized parameter control.

Preface

For silly reasons, many figures documenting in detail prototypes proposed here are omitted, but will be available on `http://folk.uio.no/rogerst/msthesis` very soon!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, at the University of Oslo, research within *active music* takes place in the collaborative research project Sensing Music–related Actions (SMA) between the Department of musicology and the Department of informatics web [2010d]. The principle goal for the SMA project is to explore *action–sound* couplings in human–computer interaction. Sub–goals include the development of technology for active music in portal media players.
Basic research questions of concern are e.g. what is the relationship between action and sound? What aspects of motion data are interesting for use in active music systems? For analysis on continuous streams of sensor data, especially relevant is the development of *machine–learning* and segmentation methods for extracting meaningful actions. Intuitively, machine learning means having machines learn by experience (i.e. increase its performance at some task), and overlaps with fields such as pattern classification and artificial intelligence Mitchell [1997].

1.1 Terms

The following are terms in need for definitions in the context of music technology.

1.1.1 Active Music

The term active music is generic and refers to music technology in which the listener can influence the music listened to. Conversely, "passive music" is more static and far less flexible for influencing it (typically, the only "musical

control" is given via buttons for pause, skip, (master) volume etc).
A top–down illustration of an active music system is given by a flowchart in Figure 1.1.

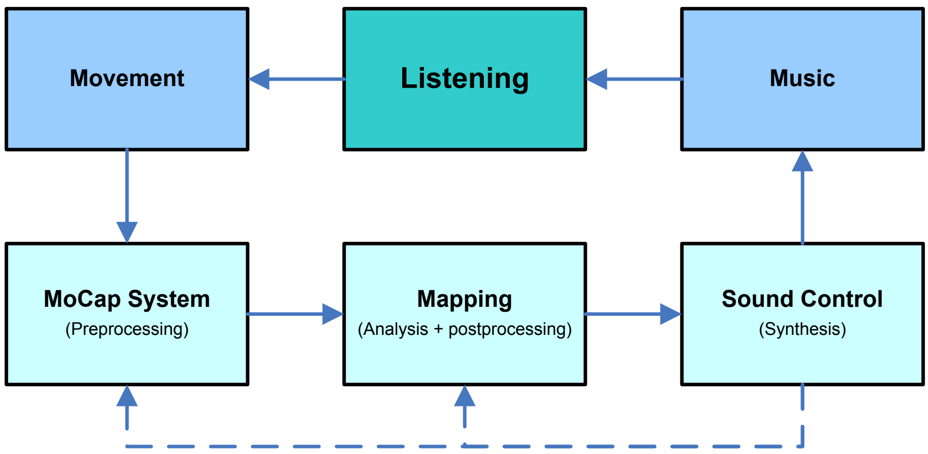

Figure 1.1: A flowchart for typical human–computer interaction in an active music system. The dashed lines refer to more advanced cases. Such a case can be analysis on combined patterns of sound features and motion features, e.g. to analyze action–sound couplings directly. Another case can be a mapping system with explicit information of states of the sound synthesis system, e.g. current tempo or current candidate pitch values that it can take regarding a given virtual instrument, etc.

1.1.2 Action–sound couplings

It is expected that there is great potential in exploiting action–sound couplings for music technology. Action–sound couplings represent relationships between actions (e.g. movements) and sound. It is believed that our life–long experiences with such couplings make us apt to imagine action or sound related to a sound–producing action that we respectively either only hear or see Jensenius [2007]. Therefore a more general understanding of action–sound couplings is considered an important basis, especially in the aid for better exploiting motion capture data for electronic active music systems. Potentially, some of these motion features can be the rhythm of the movements or the mood of the listener. Such features can then be exploited to adapt (influence) the listened music to several situations, e.g. extending the duration of a song or adapting the music tempo to one's corresponding jogging pace, or prioritize among styles and genres of the next music track according to one's present (estimated) mood Høvin et al. [2007]

1.2 Thesis overview

The main theme of this thesis is analysis of sensor data given by motion capture platforms for active music applications. Sub–systems of concern can roughly be labeled as follows;

- (a) motion capture system
- (b) mapping
- (c) synthesis

Regarding (a), especially considered sensor technologies for motion capture are wearable sensor devices that can be mounted on the body of the music listener. For instance, some of these sensor devices can constist of sensors for measuring acceleration and/or rotation in Euclidean three–dimensional space More specifically, for this thesis, a triaxial accelerometer has been chosen.

The (b) mapping system represents system– and user–controlled logic for mapping actions (e.g. sensed motions) to sound control. It includes sensor data analysis, where motions are classified (i.e. categorized) and later post-processed for sound synthesis/control Considered methods for sensor data analysis belong to the field of *machine learning*[1]. Machine learning means having machines learn by experience, often based on training examples. A more frequently cited, formal definition is cited in Section 2.2.

When it comes to (c) sound control, prototypes are implemented using the *interactive* development environment called Max [2]web [d]. Max offers good possibilities in rapid prototyping of real–time sound synthesis/processing. Also, being highly modular and relatively easy to learn, it is has more or less become the lingua franca within sound programming. Algorithms for

[1] Machine learning overlaps with fields such as pattern classification and artificial intelligence Mitchell [1997].

[2] Often referred to as Max/MSP

bottom–up sound synthesis regarding the construction of raw musical material as such are out of the scope of this thesis.

1.2.1 Prototype implementations

Two different prototypes are proposed. Both are implemented in Max/MSP and its sensor data originates from the Analog Devices' ADXL330 accelerometer adx [2007] (c.f. Figure 2.1). whose USB–to–Max/MSP interface (driver software and API for Max) is developed by Phidgets web [e].
Moreover, these prototypes have quite different application domains (the former one is more specialized than the latter). Also, they differ in their practical applicability as the first prototype yet has been significantly less successfull than the second.

LiveBot – MIDI/audio clip triggering controller for alternative music sequencing in Ableton Live

The first prototype is inspired by the concept of *hypermusic*. With hypermusic It is API–specific, and aimed at creating meta–compositions ("on the fly") in Ableton Live, a popular music sequencing program. For reasons that will become apparent, further development of this system has been put on hold (alas, at least for practical reasons, it does not yet make up for an active music application).

MaxBot – Multi–track amplitude modulator for a 7–track audio loop in Max/MSP

The second, latest prototype is an implementation of a system for continuous multi–channel amplitude control. This can be seen as a digital multi–channel mixing application. Moreover, the amplitudes are normalized so as to avoid amplification above unity (i.e. stabile control). In the implementation referred to troughout the thesis, volumes of a 7–channel audio file are modulated. The final modulation signals are generated as a function of motions and user–supplied mappings. Considering levels chained after the motion–generated modulations, the user of the GUI has the opportunity to select and configure sevaral DSP functions for different mappings/purposes. Some of these mappings, in particular based on real–time classification of motions, are programmatically interpolated.

For terminology, motions respectively transform into what I refer to as the amplitude *control* vector, and the (amplitude) *weight* vector. The weight vector is defined and interpolated by a motion classifier and a more "musically minded" postprocessor. I call these for vectors so as to include all the channels on a sample–by–sample basis.

1.3 Challenges for sensor data analysis

Frequently, a challenge in human–computer interfaces such as machine learning based active music applications, is sensor data calibration. However, a perhaps more fundamental challenge regards segmentation on the streams of sensor data.

- As humans are in constant motion, which segment lengths $\mathbb{O}$ are more "obvious" choices?
- Does the $\mathbb{O}$ of choice vary with the musical genre listened to? If so,
- - how does $\mathbb{O}$ vary with respect to musical tempo?
  - is there any universal, or culturally defined $\mathbb{O}$?

1.3.1 Data segmentation for motion analysis

Naturally, motions can be seen on multiple time–levels. However, depending on whether one wants to consider only a few of the possible durations of motion, or the *whole* range, this can lead to a practical challenge. Many classification methods require that its input (i.e. data segments) are of same size. Therefore, to be able to analyze motions on multiple time–scales can be computationally expensive[3].
Fortunately, there exist classification methods that can work on variable segment sizes. Examples include Dynamic Time Warping (DTW) wik [2011b] and Hidden Markov Models (HMMs) wik [2011c], Pylvänäinen [2005] for classification.
The classifier prototype especially considered in this thesis is based on the Support Vector Machine (SVM). Speaking for myself, SVMs do not that intuitively work on variable–width data segments, but apparently, it is able

[3] For instance, training multiple classifiers would—normally—require more processing time.

to do so Chaovalitwongse and Pardalos [2008]. It is out of the scope of this chapter (see), but ultimately, it depends on the setup of SVM[4].
However, for certain contexts of motion–capture–based musical applications, I argue that it is fair to consider only a few time–scales. As the duration of a sound–producing action often influences the resulting sound, I think it is rather plausible that arbitrary change in speed of a sound (possibly time–warped) also influences the related imaginable actions. Especially in scenarios where the active music listener wants to "fine–control" a specific sound[5], it would be natural that the motion correspond one–to–one (or few–to–one) with the resulting sound control. That is why I think it is relevant also to consider some "pseudo–synchronicity" of motion and sound for analysis. By *pseudo–synchroncity* of motion and sound, I do refer to multiple time–scales (i.e. SVM classifiers). However, —in a restricted sense, —I refer to a kind of "synchronicity" in which motion and sound relate as follows:

$$\text{MotionSpeed} \cdot 2^k = \text{SoundSpeed}, \ \ k \in \text{RestrictedSet} \subset \mathbb{Z} \tag{1.1}$$

In order to capture motions on several time–scales, an obvious—perhaps somewhat naïve—solution would be using several SVMs in parallell. Combined, these could work as a multi–level or multi–category classifier. This is not experimented with in the prototypes described in this thesis. However, classifications with different segment sizes (i.e. different datasets derived from the same acceleration stream) are explored. A less "general" solution where only one classifier is used, could be the inclusion of a few downsampled versions of maximum–sized data segments. Unless the samples are location points, they would also require some transformation in order to compensate for the sample frequency (i.e. "speed") change. Additionally, with respect to the original segment size, the reduced data segment would need to be extended (i.e. looped) so as to complete the segment. However, if the original sensor data include the constant contribution of vector amplitude such as from gravity, this is obviously not a solution. In the motion capture system applied in MaxBot, the sensor device only consist of an accelerometer. Therefore gravity's contribution is allways present in the signal. Ideally, in such a case, the noise from gravity should somehow be estimated and compensated for. For instance, without extending the sensor device with a

[4] This relates to the so–called kernel function used for training the SVM.
[5] In a broad sense; not necessarily meaning controlling a virtual instrument.

additional sensors (e.g. gyroscope), this can be done by the linear algebra gravity–correction method outlined in Pylvänäinen [2005].

1.4 Practical work

Notable practical works are as listed.

- JavaScript external development for a so–called Max for Live device (i.e. “Ableton Live external”).
- GUI and data visualization scripting in Max
- Development of Java Max externals:
  - **wml.SvmLM**: An implementation of a Support Vector Machine classifier based on a wrapper web [b] for LibSVM web [h] in the mature Weka web [g] machine learning (and pattern recognition[6]) API for Java.
  - Utilities (wml.utils):
    - **ListWindow**: A FIFO buffer for floats.
    - **RunningVoM**: Running measure of motion volume.

[6] According to wik [2011e], machine learning is a subfield of pattern recognition, which also include regression methods, i.e. predictions of a real–valued scalars or vectors —not only integers/labels.

Chapter 2

Background

Sensor data analysis for active music applications is a challenging an interesting pursuit. Active music is not a completely new area of research. It has for instance been explored with in computer games. More recent examples of commercial active music applications are e.g. RJDJ, Apple Garageband 0.9 and Microsoft Songsmith web [2010f].
Related to active music applications, and the prototypes described in the thesis, I will start this chapter with a brief description of ways for synthesizing digital music in ways that relates to digital signal processing and generation (i.e. transformation and synthesis).

2.1 Active music

Concerning the nature of a given piece of active music, one can differentiate on active music (this practically also applies to digital music in general) whose audio samples sent to the receiver (often an audio mixer) at some extremes are what can be called purely hardcoded and purely softcoded. Respectively, these labels are meant to emphasize their static (or "offline synthesized") and dynamic (or "online synthesized") nature. What is common to such active music systems, however, is the possibility to control low-level parameters, i.e. the application of DSP techniques at the sample–level (or signal–level). For example, this can result in transforming the key or tempo/duration, acoustic echo effects etc (much of which are based e.g. on (discrete variants of) the Fast Fourier Transform (FFT) for transforming a digital signal from the time-domain into the frequency-domain, and the inverse FFT transform).

2.1.1 Receiver input given solely by DSP techniques

This kind of active music is music in which the musical information source exclusively is given by a hardcoded waveform (e.g. mp3 files or an audio CD). When audio samples only from such a static waveform is given as the musical raw (input) for the receiver, transformation (or re–synthesis) of the music it represents relies solely on the application of digital signal processing (DSP) techniques (e.g. such as amplitude or frequency modulation, granular/grain (re)synthesis etc.).

2.1.2 Receiver input given both by DSP and *DSG* techniques

The second kind is music can be seen as an extension of the former. For the receiver, the waveform input (at least if thought about on a larger time–scale), – besides most often also given by DSP techniques, – is given by digital sound *generating* (DSG) techniques. This is music that is programmable along many more dimensions. For example it is possible to control individual sounds separately, and manipulate contents of the musical piece at higher levels of abstraction. Hence both high-level musical parameters (e.g. tempo, key ...), mid-level parameters (relating to e.g. virtual music instruments, sound effects or musical scores), and low–level parameters at the sample–level, are programmable. Obviously, this makes it possible to influence the (interactive) music at a much larger extent than working solely on sample-level with a (pre-synthesized) waveform file. Examples include the possibility to create remixes or alternate compositional versions "on the fly". An example of music technology for such interactive music capable of the latter is called hypermusic [Høvin et al., 2007, cited Machover [2004]]. This is a technology under research and development in the SMA project, and especially, it also is the basis behind the development of the projected portable active music player.

2.1.3 Relevant technologies and tools

Motion capture technology is often a natural (intuitive) basis for active music systems.
There are quite a lot of sensor devices relevant for different contexts. Some sensors measure biosignals (such as e.g. muscle contractions (EMG) or elec-

troencephalogram (EEG) for measuring brain activity by means of electrodes placed on the scalp), others are e.g. force-sensitive resistors, light sensors, microphones, capacitive sensors for measuring distance, etc. web [f] However, for measuring movements, possibly optical and on–body kinematic/inertial sensors are more relevant.

Optical Sensors

Today, frequently for practical purposes, a relatively common choice of sensor devices for motion capture is ordinary video cameras. These are usually quite easy to work with, though relatively processor intensive —typically with millions of pixels to monitor for relatively few interesting tracking points. Not too long ago, Microsoft announced their Kinect 3D motion capture (multi–sensor–based) device for Xbox wik [2011d]. Such technology seems promising, at least for budget class 3D motion capture technology. For instance, a somewhat older technology such as *stereoscopic vision* wik [2011f] adds up to the computational intensity in that tracking requires a setup of multiple video cameras. Even then, (although at a smaller degree,) possible occlusion by objects in front of a camera can make it impossible to obtain continuous 3D tracking. An other type of video-based 3D tracking involves using multiple infrared-sensitive cameras Nymoen [2007]. Such equipment is e.g. used for animation purposes, but are also quite expensive today.
A common practical downside for video–based tracking is that only the quite expensive ones fulfill high requirements for latency, spatial and temporal resolution (e.g. frame rates) for modern real-time motion capture based musical interfaces. Typically, when affordable cameras fulfill a desired temporal resolution, they lack the desired spatial resolution, or vice versa.

Motion Sensors

The more recent possibility of using small sensor devices that are implemented with MEMS[1]–based integrated circuits offer advantages. Such sensor devices are relatively energy–efficient, typically affordable, and small enough to fit into light–weight containers that can be placed on body parts. Examples of popular types of motion sensors are *inertial measurement units* (IMUs). IMUs combine accelerometers (e.g. Analog Devices' ADXL330 adx

[1]Micro-ElectroMechanical Systems

[2007], Figure 2.1) and gyroscopes measuring rotational velocity for 3D relative positonal tracking (e.g. used in navigation systems). This has also already been used in commercial products, such as the Nintendo Wii remote controller, Apple's iPhone, and products from Xsense. However, a downside especially for gyroscopes is drift (i.e. linear noise) in their voltage output.

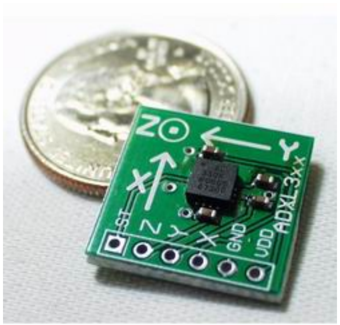

Figure 2.1: The ADXL330 accelerometer MEMS chip from Analog Devices.

2.2 Machine Learning

For sensor data analysis, machine learning techniques have shown to be a promising toolbox. This is an inter–disciplinary field concerned with algorithms that automatically make a computer program's performance improve with experience. A commonly cited definition of machine learning is given by Tom M. Mitchell and goes as follows

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

There are a Machine learning typically[2] involves "adjusting" the machine's internal model based on training data in order to "predict" information about future input patterns so as to optimize accuracy or fitness[3] function.

[2] Sometimes, it can be more appropriate to speak of e.g. searching, evolving or optimizing.

[3] For instance, within the theory of evolutionary algorithms, *fitness function* is a common name for a function measuring performance of a genome (i.e. a candidate model evolved by some selection and/or mutation mechanism) Eiben and Smith [2008].

Classifiers are common machine learning systems. Preprocessing and feature extraction are often embedded in classification systems as they can be performance–increasing. Typical preprocessing can be data segmentation and noise removal (e.g. eliminate information from irrelevant transformations). A general outline of a classifier is illustrated in Figure 2.2.

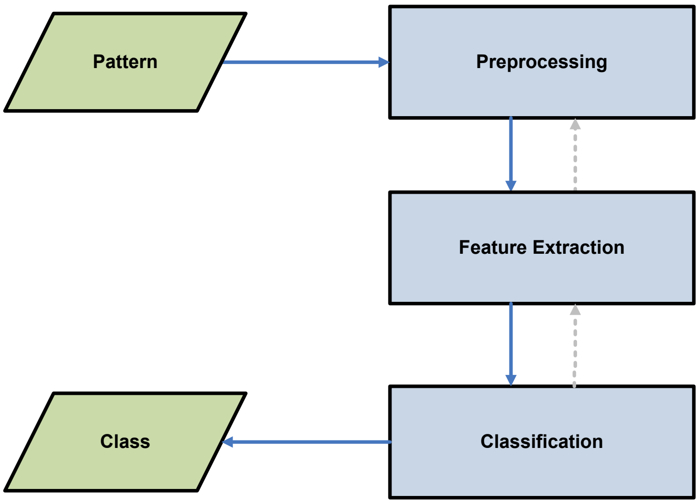

Figure 2.2: General outline of a classication system.

Some common machine learning methods are *kernel methods* (KMs) — with the Support Vector Machine as its most known family member, — *artificial neural networks* (ANNs) and *evolutionary computation* (EC). Both of the latter are inspired by and draw on concepts from biology.
Evolutionary computation (EC) is concerned with search algorithms inspired by biological evolution. such as selection, recombination and mutation. In an implementation of an evolutionary algorithm (EA), given proper parameter values for the EA, it will often find quite good or (near) optimal solutions faster than other approaches.
In particular ANNs are inspired by neurobiology, with emphasis on relations between neurons or —occasionally—algorithmic aspects of brain areas, e.g in so–called Hierarchical Temporal Memory Hawkins and Dileep [2007]. ANNs must be trained on example data and learn by gradually changing the weights between pairs of ANN nodes. A classic training method is Backpropagation which is a method based on gradient descent.
The network topology of ANNs can be arbitrary, but in common they all

have three types of layers of nodes that represent multiple neurons. The corresponding layers are the sensing layer for the network inputs, one or more hidden layers, and a top level for the network output(s). Figure 2.3 illustrates a (minimized) ANN topology. A pioneering ANN learning method is called Backpropagation, it was

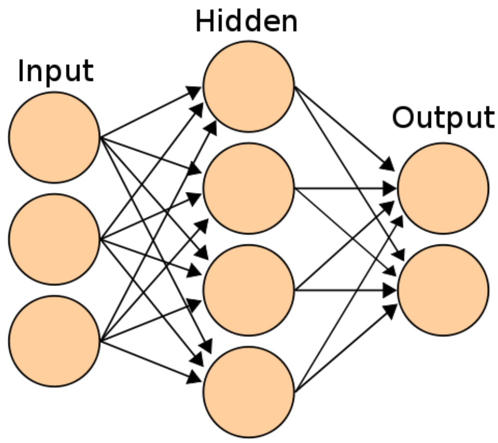

Figure 2.3: Concepts of an artificial neural network with a basic set of layers (i.e. one hidden layer, plus the I/O layers). Image found in wik [2011a].

Apart from training an ANN on adequate example data so as to better generalize on unseen examples, another challenge is to find good network topologies for the hidden layer. Using machine learning techniques based on methods mentioned above, it is possible to automatically (re–)model (learn from experience) and/or evolve network topology for performance-improving relations between input and output. Moreover, a phenomenon often occuring when training an ANN is *overfitting*. This happens when the ANN is been tuned to capture information about its example data that is badly representable for yet unseen examples.

2.2.1 Classification with Support Vector Machines

Support Vector Machine (SVM) is a popular supervised learning method designed for classifying patterns represented by real vectors. This learning method is also designed to avoid overfitting problems, i.e. it often generalizes (learn) quite well. Per se, it is designed for binary classification. However,

methods for transforming multi–class classification problems into multiple (binary) SVM classification problems do exist.
The goal of SVM is to find an optimal hyperplane that separates the two classes of patterns by having the largest possible *margin*. The margin is simply the geometric (Euclidean) distance from the hyperplane to the nearest patterns, respectively from the first and second class (both half–spaces defined by the hyperplane). These (nearest) patterns (from each of the two classes) represent patterns that are the most difficult to classify (correctly), and are called *support vectors*. Support vectors have the same distance/margin to the hyperplane, and form the basis for the hyperplane which then can be called a maximum–margin hyperplane. It is expected that the larger this margin is, the better the classifier will generalize (beoynd seen patterns from the training set). This maximum–margin hyperplane (classifier) is illustrated in Figure 2.4[4].

SVMs are designed to work with linearly independent training sets, there-

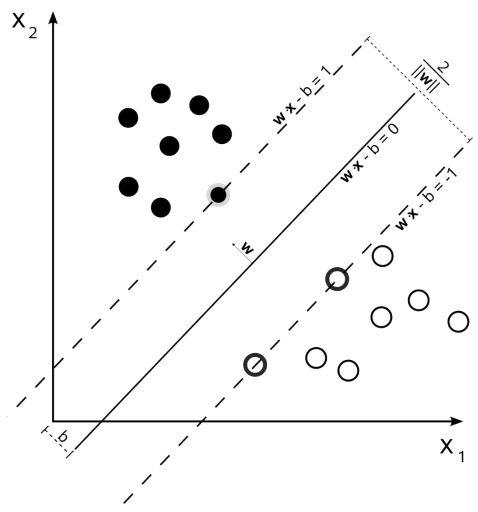

Figure 2.4: Example of a maximum–margin hyperplane (in feature space) obtained by training an SVM. This separating hyperplane is situated where $\mathbf{w} \cdot \mathbf{x} - b = 0$. The support vectors are those intersecting the "support hyperplanes" at $\mathbf{w} \cdot \mathbf{x} - b = \pm 1$.

fore quite often the training set requires preprocessing. Fortunately, when the training set in question is not linearly independent (in its original vector space), it can (virtually) become linearly independent [Duda et al., 2000, p.

[4]Image found in wik [2010a]

259] by applying an adequate nonlinear mapping $\boldsymbol{\varphi}(\,\cdot\,)$ on the orignal vectors from the training set onto a space of a sufficiently higher (sometimes even infinite) dimension. To actually find such a mapping in practice can be tricky, however there exists methods for minimizing the classification error. A geometric illustration of the concept of such a mapping is given in Figure 2.5[5].

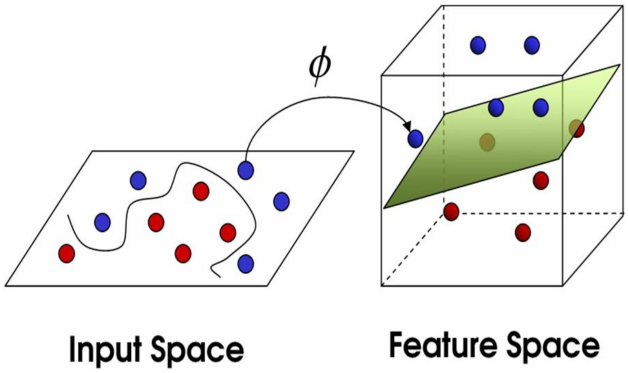

Figure 2.5: An illustration of the goal of SVM, which is to find an adequate mapping $\boldsymbol{\varphi}$ (vector function) that transforms linearly dependent vectors into linearly separable vectors in a space of higher dimension (hence the hyperplane). The nonlinear decision boundary in input space is found after SVM training.

Training an SVM

Assume that initially we have a training set $\mathcal{X}$ consisting of k linearly non–separable vectors (patterns) $\{\mathbf{x}_i\}_{i=1}^k \subset \mathbb{R}^m$. We denote the associated classes by $\big\{t_i \in \{-1,1\}\big\}_{i=1}^k$. Then we let a transformed training set $\mathcal{Y}$ consist of the linearly separable (independent) vectors $\{\mathbf{y}_i\}_{i=1}^k \subset \mathbb{R}^n$, where $n > m$, be defined by an adequate mapping $\mathbf{y}_i = \boldsymbol{\varphi}(\mathbf{x}_i)$. Here $\mathbb{R}^m$ and $\mathbb{R}^n$ are respectively referred to as *input space* and *feature space*. More formally, we define this by

$$\mathcal{Y} = \big\{(\mathbf{y}_i, t_i) \mid \mathbf{y}_i = \boldsymbol{\varphi}(\mathbf{x}_i) \in \mathbb{R}^n,\ \mathbf{x}_i \in \mathbb{R}^m,\ n > m,\ t_i \in \{-1,1\}\big\}_{i=1}^k,$$

in which each element belongs to either one of the classes ω_1 and ω_2. We let the class–belongings to these vectors be mapped by

$$t_i = \begin{cases} 1 & \text{if } \mathbf{y}_i \text{ belongs to } \omega_1 \\ -1 & \text{if } \mathbf{y}_i \text{ belongs to } \omega_2 \end{cases}, \quad \forall\, i \in \{1,\ldots,k\}.$$

[5]Image found in Gisler [2008]

Now we can start finding the hyperplane. From linear algebra we have that any hyperplane $\mathcal{H}$ can be expressed as

$$\mathcal{H} = \{\mathbf{x} \mid \mathbf{w}\cdot\mathbf{x} + w_0 = 0\} = \left\{\mathbf{x} \;\mid\; w_0 + \sum_{i=1}^{m} w_i x_i = 0 \;:\; \mathbf{x}, \mathbf{w} \in \mathbb{R}^m\right\}.$$

In order to re–express this condition ($\mathbf{w}\cdot\mathbf{x} + w_0 = 0$) to a more compact, homogenous equation on the form

$$\mathbf{a}\cdot\mathbf{y} = 0,$$

we can let the weight–vector $\mathbf{a}$ and the feature–vector $\mathbf{y}$ be augmented versions of $\mathbf{w} = [w_1 \dots w_n]^{\mathrm{T}}$ and $\mathbf{x} = [x_1 \dots x_n]^{\mathrm{T}}$ respectively. by

$$\mathbf{a} = \begin{bmatrix} w_0 \\ \mathbf{w} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}.$$

Now, we can say that

$$g(\mathbf{y}) = \mathbf{a}\cdot\mathbf{y}$$

is a linear discriminant, and test vectors are classified according to the sign of $g(\mathbf{y})$.
The corresponding hyperplane (given by $g(\mathbf{y}) = 0$) we are looking for then ensures that

$$t_i g(\mathbf{y}_i) \geq 1, \quad \forall\, i \in \{1, \dots, k\}. \tag{2.1}$$

(The subset of transformed feature vectors $\{\mathbf{y}_i\}_{i=1}^{k}$ that gives equality in (2.1) are namely those called support vectors.)
Further, since the distance from a hyperplane $\mathcal{H}$ to a transformed feature vector $\mathbf{y}$ can be shown to be $\frac{|g(\mathbf{y})|}{\|\mathbf{a}\|}$, which implies that

$$\frac{t_i g(\mathbf{y}_i)}{\|\mathbf{a}\|} \geq b, \quad \forall\, i \in \{1, \dots, k\}, \tag{2.2}$$

where b is the margin. Now, (2.1) and (2.2) imply

$$b\|\mathbf{a}\| = 1, \tag{2.3}$$

and the goal then becomes to find the weight vector $\mathbf{a}$ that maximizes the margin b.

This optimization problem can be formulated by the method of Lagrange undetermined multipliers, in which we want to minimize $\|\mathbf{a}\|$, i.e. the norm (or length) of $\mathbf{a}$. Because this method involves derivation, one can simplify the algebra by solving the equivalent problem of minimizing $\frac{1}{2}\|\mathbf{a}\|^2$. With respect to $\mathbf{a}$, one wants to minimize $L := L_{min}$, defined by

$$L_{min}(\mathbf{a}, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{a}\|^2 - \sum_{i=1}^{k} \alpha_i \left[t_i g(\mathbf{y}_i) - 1\right], \ \forall\ \alpha_i \geq 0, \tag{2.4}$$

and maximize it with respect to the undetermined multipliers $\{\alpha_i \geq 0\}_{i=1}^{k}$. However, it can be shown that by using a so–called Kuhn–Tucker construction [Duda et al., 2000, p. 263], this can be reduced and re–expressed purely as a maximization problem. We refer to this problem with L_{max}. L_{min} also takes $\mathbf{a}$ into account, however, the Kuhn–Tucker construction depends only on $\boldsymbol{\alpha}$, which is defined by

$$L_{max}(\boldsymbol{\alpha}) = \sum_{i=1}^{k} \alpha_i - \frac{1}{2}\sum_{i,j}^{k} \alpha_i \alpha_j t_i t_j \left(\mathbf{y}_i \cdot \mathbf{y}_j\right), \tag{2.5}$$

subject to the constraint

$$\sum_{i=1}^{k} t_i \alpha_i = 0, \quad \alpha_i \geq 0, \quad \forall\ i, j \in \{1, \ldots, k\}. \tag{2.6}$$

When $\boldsymbol{\alpha}$ is found by using (2.5) and (2.6), one can combine the answer with (2.4) and find $\mathbf{a}$. The margin b is given by (2.2), i.e. $b = \|\mathbf{a}\|^{-1}$. There are multiple methods for solving (2.5) with the condition (2.6), one is called quadratic programming.

Chapter 3

Implementations

3.1 Motion Data Analysis

In this thesis, the sensor data mapping is based on machine learning theory, in which its fundamental sensor data analysis is based on pattern classification theory (i.e. these aspects very much overlap with respect to analysis). Intuitively, we want to have gestures classified (analysis–related), and somehow use the class–predictions for sound control (mapping–related). Gestural information represented by time–series of sensor data are obviously more or less hidden in the sub–intervals of the sensor data stream. Also, the gestures (whose sensor data aspect is represented on sub–intervals) may overlap with other gestures.

3.1.1 Motion Capture Platform (Server)

On the server–side, sensor data are transmitted over USB from a three–dimensional ADXL330 accelerometer – and received (sample by sample) in Max. The system is thus directly connected with the accelerometer via Phidgets driver and interface for Max, and provides both for the bypassing of raw (albeit calibrated) acceleration samples, and for the computation of various features/transformations from the raw acceleration signals.
It was earlier planned to use these features for classification, although it does not yet seem necessary for successfull motion classification.

In this prototype, these "features" are not used for analysis (at least not yet). However, they are applied for 7–channel amplitude synthesis, or amplitude envelope synthesis. Actually, the user can choose to have them bypassed to the client, or via an amplitude envelope follower/synthesis filter. The input for the classifier is the sampled (time) series of a three–dimensional real vector (window length is constant). The overall data flow in this subsystem is illustrated in Figure 3.1 Its implementation in Max is illustrated in Figure 3.2.

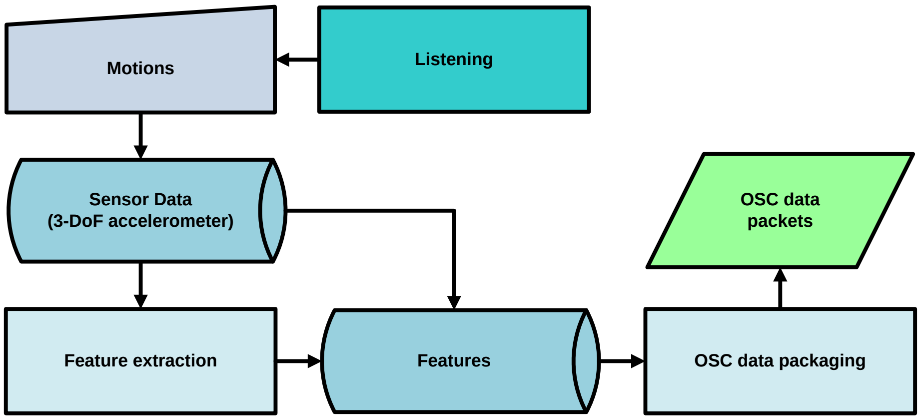

Figure 3.1: Data flow in the sub–system for realtime extraction of features from sensor data.

3.1.2 Perceiving Musical Motions

In this thesis, gestural information is represented in overlapping time–windows of fixed length. The classification of a gesture is therefore sensitive to its speed. This is definitely not always desirable, but in the context of musical performance, I think it is not that far–fetched to somehow take tempo into account. Especially in order to artificially perceive the same gesture on different musically compatible time–scales, it is of course possible to transform the acceleration data segment into alternative undersampled versions of the original gesture. This, however, has implications for aliasing the signal in the frequency domain. Moreover, as the speed of the same motion varies linearly, the acceleration amplitudes varies nonlinearly. It is not intuitive how these amplitudes vary with respect to speed, especially when a constant gravity is part of the signal. Also, as regards to the classification method especially

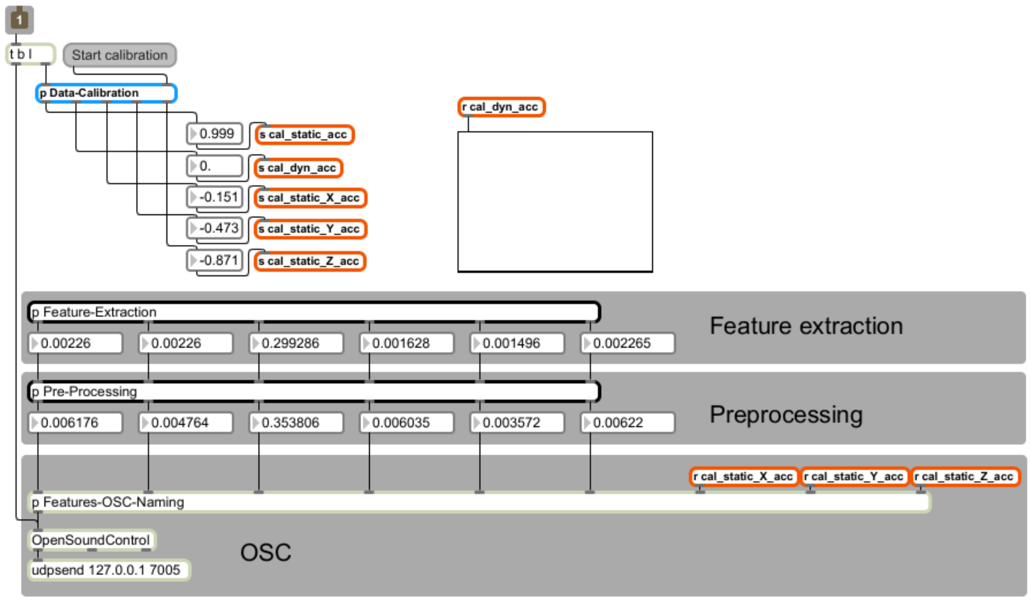

Figure 3.2: Max patcher implementation of the server subsystem for calibration on the acceleration data and transformation to 7–channel control amplitudes (on client–side modulated by a running interpolation of preset weight vectors).

considered in this thesis (Support Vector Machine), the time–series to be classified must have the same dimension.

It is therefore an open question whether the series of gesture data as perceived through a fixed–length time–window should be conceived of as being contiguously defined or defined in sub–sequences. In this thesis, however, "gestures" are narrowly conceived in terms of any fixed–length (time–)series of acceleration samples (represented in a fixed order with respect to the time of sampling).

What is questionable with regards to how one defines a gesture, is to what extent is a gesture a series of acceleration samples can represent gesture is an open question, however However, before speaking of classifying motions, somehow, it should be expressed how we want to represent motions.
What we have is a series of sampled data points of 3D accelerations (i.e. time–series). Then, what are the adequate ways of representing motions based on this input? An intuitive choice is to add the subsequent sampled acceleration points into a buffer which then represent the accelerations sampled within a given time–scale (given by the buffer's size). Adding the accelerations into this buffer should also be in a fixed order, and for instance let the

order correspond to the time of sampling. Such a time–series (signal) then implicitly represent motions sampled within some window of time. As shown in the chapter on experiments, such a choice is indeed adequate.

3.2 Mapping Systems

Two different mapping systems for different application domains are developed – to different extents. Both systems are developed in Max (with different sets of mentioned externals), however, one of these is more specialized towards applications with the Ableton Live music sequencer. In common, they are based on the same motion capture platform. This motion capture platform is based on Phidgets' USB interface for a wearable accelerometer, but they differ in the application domain, i.e. having different sound control clients.

These systems use the Open Sound Control data communication protocol (OSC, a UDP abstraction) for server–client communication. This makes the system more modular since it can also communicate with any OSC–compatible client (i.e. not only Max), e.g. Ableton Live. Both mapping systems are thus twofold and implemented in Max with the use of first– and third–party externals (extensions for Max).

3.2.1 LiveBot

Regarding sound control, the first prototype is aimed at discrete auto–triggering/playing of MIDI/audio clips in a multi–track digital music sequencing software [1] from Ableton named Live[2] web [a]. At the time of developing this prototype, I assumed that information from the Live API web [c] (for Max) about these clips' start– and endpoint from the linear musical arrangement view (which represents the precomposed static clip sequencing composition meant to be virtually altered (in real–time)) was available, but such information lacked totally. Therefore, in order to actually implement any automatic alternative clip triggering, this kind of information somehow had to be hardcoded. First, I added this information manually into each clip's name—a time–consuming and error–prone process.

This prototype has not fully been implemented. This API–related issue,

[1] Often referred to as a digital audio workstation (DAW) software wik.

[2] Often referred to as Ableton Live

made testing and development an error-proned and time–consuming process. Therefore further development has been aborted. This system is inspired by the concept of hypermusic but implemented only partially. It is a bit complex to explain in how, but as an "existence proof", using a simple, albeit manual and error–prone "clip labelling" approach (exactly what approach will become clear), it has been demonstrated that it is possible to recreate the original (MIDI/audio) clip playing sequence which again,—although abstractly— hints that exchanging playback of original clips with new compatible ones is indeed possible. However, for technical and practical reasons, further implementation has been put on hold.

3.2.2 MaxBot

The second – and latest – prototype is more general–purpose in nature. It is made for continuous multi–track amplitude modulation, and is here applied for volume mixing on a 7–channel audio file. Mathematically, its output is a vector whose elements vary in the $[0, 1]$ range. Therefore, by simply extending the prototype for instance with a UDP (or OSC) server for data communication, it can virtually be applied to any situation requiring non–amplifying amplitude modulation.

For an overview of the machine learning (sub–)system, see Figure 3.4.

This client–side application of the motion capture platform receives both the raw acceleration vectors and (derived) features (extracted in the server subsystem). The client should perhaps compute these features in order to off–load network traffic, and be more scalable, but this not a major issue (this is merely a prototype, but worth the note). In essence put, (the final) channel amplitudes/volumes are controlled by multiplying the feature vectors with the resulting weight vector from a running linear interpolation ("cross–fading") between pairs of user–defined (or preset) vectors. The loading of new (preset) vectors to perform interpolation on can be controlled by the user, or alternatively controlled by a learning machine (e.g. as a function of the learning machine's series of recently predicted gesture labels). All channel amplitudes (represented by vector elements) take real (float) values in the $[0, 1]$ range, i.e. it does not increase the original channel amplitudes. The mapping of features to the multi–channel amplitude vector is illustrated in Figure 3.3.

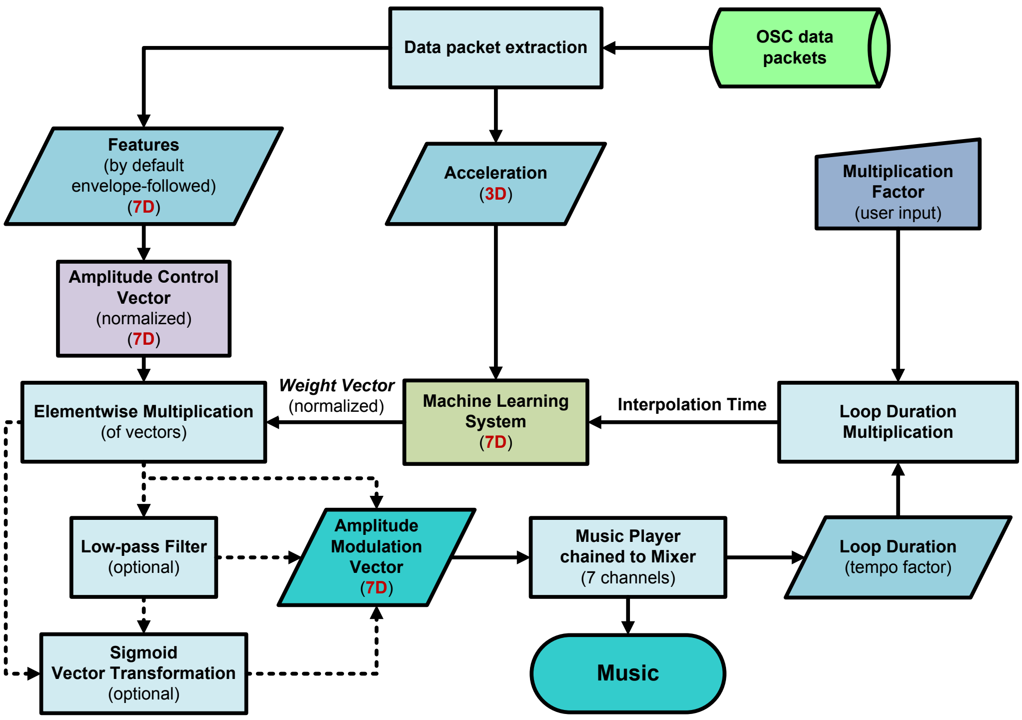

Figure 3.3: A flowchart for the (OSC) client–side "MaxBot" implementation of a 7–channel amplitude control system. The machine learning system is a sub–process which is expanded for illustration in Figure 3.4. NB: Here the dotted lines represent exclusive output directions (similar to subclass arrows in UML).

3.2.3 Mapping acceleration data to multi–channel AM synthesis

Beyond the actual feature extraction (separate patcher for this), the main patcher (menu) for the system is illustrated in Figure 3.9. Visualization of sensor data features (or, the feature–vector) can be viewed in Max patchers as illustrated in Figure 3.5 where linear interpolation is enabled, and in Figure 3.6 where nonlinear ("sigmoidal") interpolation occurs.

From the accelerometer user's perspective it is, – besides turning off automatic control and manually adjusting the master volume vector, – possible to control the volume vector on two levels. What controls the weight–vector depends on if weight–vector interpolation is enabled or not. If the interpolation is disabled, the weight vector is directly controlled by the the red sliders illustrated in Figure 3.10. If weight–vector interpolation is enabled,

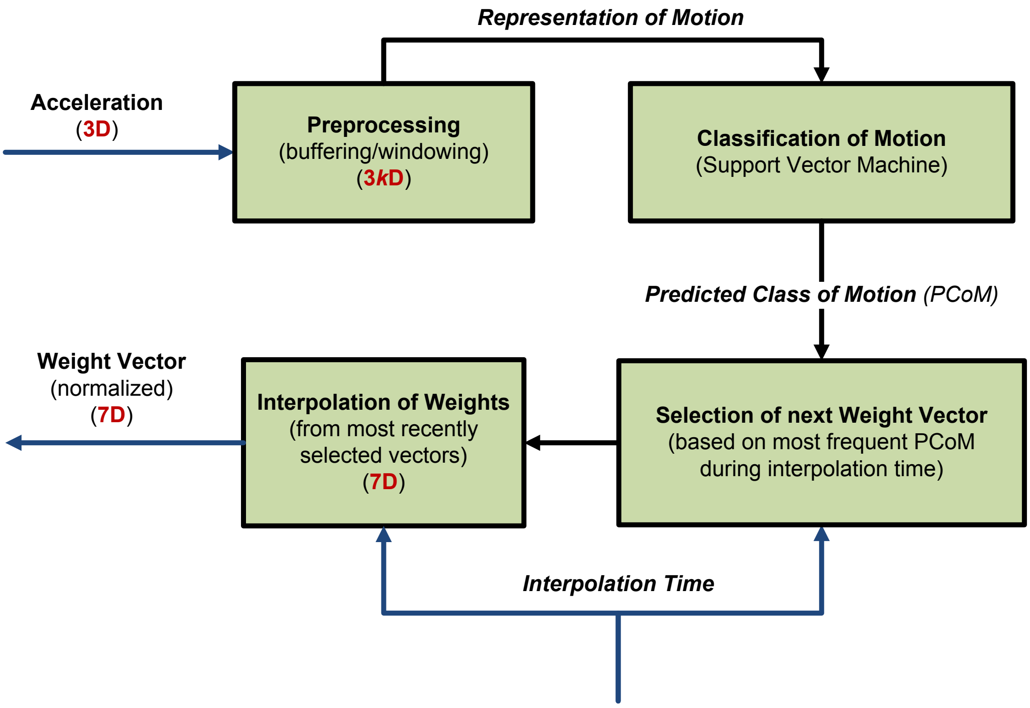

Figure 3.4: NB: Here the blue lines represent the input and output for the sub–process (the surrounding flow is illustrated in Figure 3.3).

the resulting interpolated weight–vector is illustrated by the green sliders in Figure 3.7. The interpolation interval (speed) can be set in the Max patcher illustrated in Figure 3.8.

Thus, (main) user–controllable aspects are as follows:

1. Set/reset (or disable/pause interpolation of) the weight vector, and control the volume vector (only) as a function of the amplitude control vector (i.e. the possibly ADSR–filtered amplitude control signal).

2. Let the weight vector be automatically controlled/interpolated (by the learning machine), and let the final volume vector be controlled/updated as a function of this weight vector and the feature vector.

3. Define normalized linear (scaling and bias) transformation of channel volumes with simple sliders (colored in green in ??).

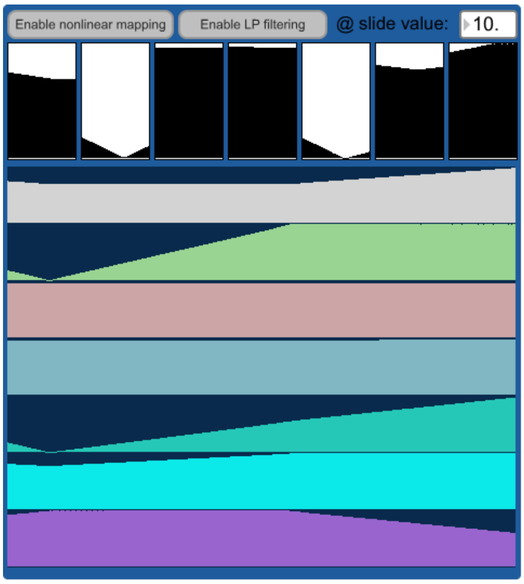

Figure 3.5: Max patcher for the (client–side) weight–vector interpolator (presentation mode). Here, the interpolation is linear (default).

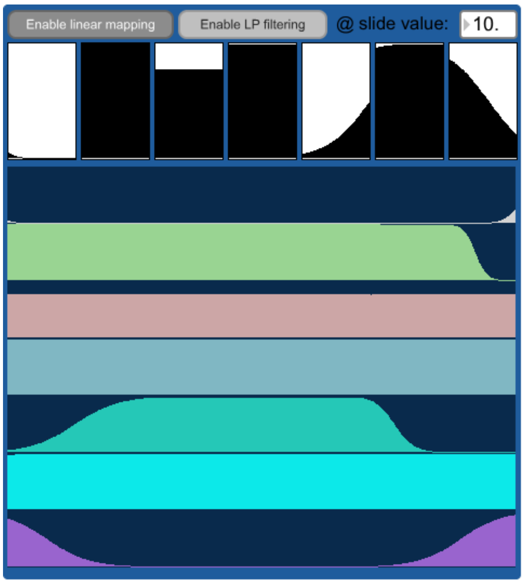

Figure 3.6: Max patcher for the (client–side) weight–vector interpolator (presentation mode). Here, the interpolation is nonlinear ("sigmoidal").

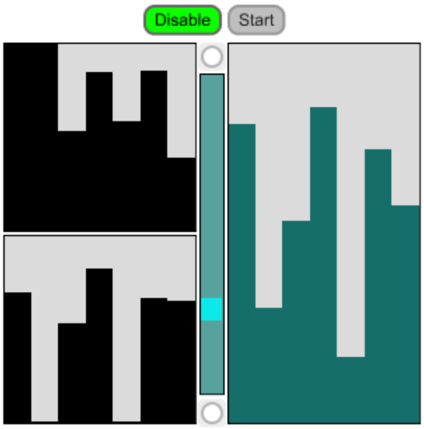

Figure 3.7: Max patcher for the client–side weight–vector interpolator (presentation mode).

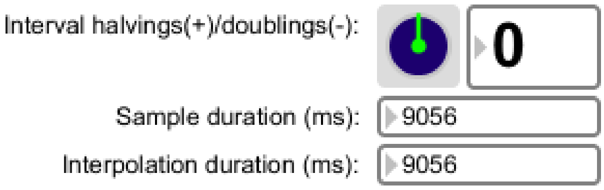

Figure 3.8: Max patcher (client–side) for controlling the interval of the weight–vector interpolation (presentation mode).

Figure 3.9: Max patcher for the client–side system menu (presentation mode).

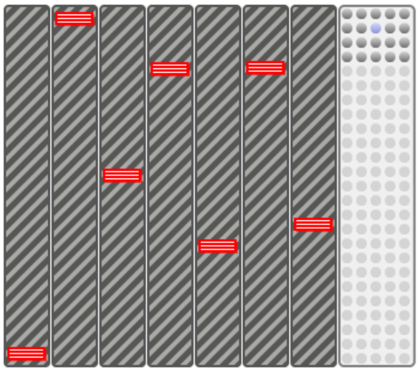

Figure 3.10: Max patcher for defining (and storing as presets) the available weight–vectors.

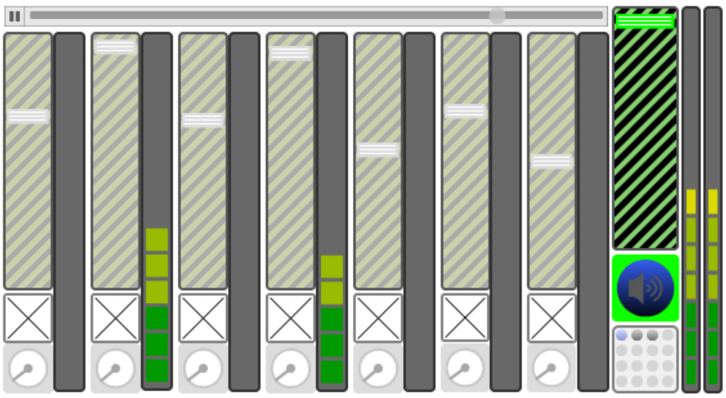

Figure 3.11: Max patcher for the multitrack audio player and mixer (presentation mode). Column–wise, the sliders determine the respective channel volumes.

Client–side mapping

Analysis of gestural data In brief terms, captured gestural data are transformed into AM synthesis, controlled by a classifier–based, supervised learning machine.

Representation and preprocessing of motion data A discrete lossless representation of acceleration–sensed motion is here represented by the contiguous historical series of the accelerometer samples, i.e so–called time–series data. More specifically, before analyzing these time–series, in order to obtain data over a given time period, each sample–vector is added into a first–in–first–out (FIFO) buffer (i.e. "stream buffer" of a constant size). Then, at some n–th time–step, the buffer's data (i.e. a $3k$–dimensional contiguous (historical) part of the acceleration signal) is sent to the classifier. If the classifier already has been trained on some (labeled) data), it's output is the predicted label associated with the (windowed) acceleration signal.

Classification of gestural data In the literature, at least for time–series *regression* (prediction of a real number/vector) one wants to learn/approximate some function

$$f(\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots \mathbf{x}_{n-k}) = \mathbf{x}_{n+1}$$

, i.e. "predict" the future/next input–vector (given a (historical) time–series), the radial basis function (RBF) is often considered a good kernel function candidate. Therefore, intuitively, since in fact the classifier in this prototype operates on input–vectors (implicitly) representing time–series (i.e. series of data captured over time), – for me – it is natural to consider classifier performance using the RBF kernel. It seems that software such as e.g. Wekinator (based on the Weka machine learning library), feature common kernel functions (e.g. RBF, linear, polynomial...), but as I have a time–limit on my master's project, I have considered it "risky practice" to learn how to use (and possibly "hack" – which anyway I had to do in the beginning, to make it work on my Windows computer) this software within the given amount of time, and less risky to develop a Max Java external of an SVM learning machine based on Weka. To my frustration, however, I ended up using a great deal of time on this "Weka SVM for Max" project of mine anyway, but finally, now it works. It is a simple classifier, but has what I was looking for, namely the ability to configure the kernel function (among a few other parameters) and

save/load the classification model ("learning machine knowledge").
The classifier in this system is a Java external implementation based on the Weka web [g] (a mature machine learning API for Java) Java wrapper for LibSVM web [h], which is an implementation of the famous machine learning method named Support Vector Machine wik [2010a]. The input for this external is a Max list of floats (representing a real 3–dimensional vector) of dimension 3 (although one can change this by sending it messages/arguments about the input list size ("dimSize") and its internal window length ("windowSize")). Depending on the training status of the classifier, the input may also be shipped with a class label. Therefore, – disregarding the possibly present class label, – the actual input for the classifier used in this system is a $3k$–dimensional window of the (calibrated) raw 3–dimensional acceleration samples (acceleration patterns over multiple time–steps) captured from the accelerometer. During (batch) training, the (supervised) learning machine in this system, "learns" as a result of forming an adequate internal label–prediction (classification) model, i.e. from the set of constant–dimensional data perceived through its given (often quite limited, but hopefully representative) set of (vector, label) examples. After the learning machine (hopefully) has formed some adequate knowledge of its world, i.e. in its "post–trained" operating mode, the input for the learning machine's classifier is simply the (calibrated) raw 3–dimensional acceleration samples, (post–)processed into windowed ($3k$–dimensional) time–series data (i.e. a digital signal).

Behaviour of the learning machine (synthesis) Like most learning machines, its prediction controls some action/behavior. In this system, briefly put, the behaviour of the learning machine is the control of a 7–dimensional weight vector that is element–wisely multiplied on the 7–channel amplitudes, which in its turn is updated as a separate function of the accelerometer data. The learning machine's behaviour, is, at the top level, implemented by a linear interpolation over two weight–vectors. When the interpolation factor is 1 and 0 (at the boundaries), the weight–vector that is multiplied with 0 is replaced by a new one. And, at the end of the chain, the user can also choose between no further mapping (i.e. keeping it linear) and a nonlinear sigmoid mapping.
Regardless, the weight–vectors are element–wisely multiplied [3] with the feature–

[3] It seems there does not exists any common mathematical operator for element–wise vector multiplication web [2010e], however, for $n \times 1$ vectors $\mathbf{a}, \mathbf{b}$, the operation is equivalent

vectors. Selections of these pairs of vectors are determined as a function of the classifications that have occurred over the past two interpolation periods. This learning machine determines the next weight–vector to interpolate onto (i.e. multiply/amplify from 0 to 1) as a function of the most frequent label classified (mfl). When the learning machine is not yet trained or simply disabled (i.e. not performing classifications), this weight vector, – say **b**, – is constant and set to $\mathbf{1}_7 = [1111111]^{\mathrm{T}}$. In this case, in other words, it does not transform the ADSR feature vector **s** to a different one as it normally would (either by the desktop user or the learning machine). As for now, two–category classification is performed. To add some variation, by design, the selected weight–vector is randomly drawn from two exclusive subsets of the pool of all preset weight–vectors (e.g. presets indexing from 1 to 10, and 11 to 20). The interpolation periods are by default set to the duration of the looping audio file, although the user can (and probably should) adjust/vary the the number of doublings or halvings of the interpolation period (set to 0 by default). In other words, for an audio loop lasting 2^n beats, the interpolation duration is drawn from a small subset of "compatible" tempos relative to the duration of the (looping) audio file. Thus, mathematically, the interpolation interval (loop) can be expressed as lasting for $2^k \cdot 2^n = 2^{k+n}$ beats. Many other interpolation intervals could be available for the user (e.g. $1/3, 1/6$), but I think – at least for starters – this is a minimal set of musically fool–proof interpolation intervals. Weight vectors as such is thus defined by the user, regardless of movements, while the resulting interpolated weight–vector is determined as a function of the gestures (classifications).

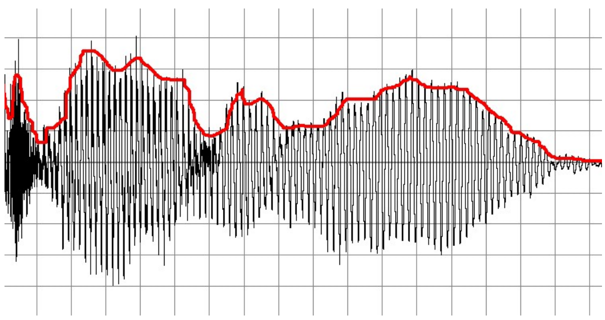

Figure 3.12: Here, the red curve illustrates "envelope-following" for an input signal (in black). Image found on wik [2010b].

to $\mathrm{diag}(\mathbf{a})\mathbf{b}$.

3.3 Third–party externals overview

The following third–party externals used in these systems (LiveBot and MaxBot) are note–worthy:

- Externals from Phidgets for accelerometer–USB interface (sensor data sampling)
- **smoother** [4] which is based on envelope–following wik [2010b] (DSP filter) whose principle is illustrated in Figure 3.12. In MaxBot, it serves as a low–pass filter for preprocessing the amplitude control vector signal generated by the sensor data. Moreover, I find its effect to be very similar to the Attack–Decay–Sustain–Release (ADSR) filter commonly used in digital musical instruments (e.g. such as sound synthesizers) filter for amplitude modulation in the time domain. This is a common component of many virtual instruments.
  Simply put; for any input sample of larger amplitude than the previous sample, the envelope–follower filter **smoother** produces a series that begins at this local peak and smoothly decreases in value—e.g. quite similar to what happens when you hit a piano key
- OSC externals from CNMAT's Max/MSP/Jitter depot [5].
- **ej.linterp** Java external for list interpolation, made by Emmanuel Jourdan [6]. Applied for interpolation between presets of so–called weight–vectors (active (interpolated) presets are determined as a function of the classifier's last label–outputs).

[4] External developed by Ph.D. Tristan Jehan at the Massachusetts Institute of Technology: `http://web.media.mit.edu/~tristan/maxmsp.html`

[5] The "Everything for Windows" pack, dated 2011/04/04, at `http://cnmat.berkeley.edu/downloads`

[6] `http://www.e--j.com/?page_id=165`

Chapter 4

Experiments

4.1 Classification experiments

The following are two sets of classification experiments that illuminate the (expected) lacking effect for varying the window (i.e. segment) sizes used in a sliding window method for motion capture. The step size for the sliding window is 1. In common, the results from these sets of experiments measure accuracy, which is the number of correctly classified instances relative to all instances. The first set of experiments also measure class precision and class recall. Respectively, these measure the true positive rate and the false negative rate for the class in question.

4.1.1 A few experiments of the effect of window segmentations on a large two–category dataset

The following subsubsections show results from classification experiments evaluated with a 5–fold[1] crossvalidation. The dataset is equally balanced and based on the same two streams ("superclasses") of triaxial acceleration samples (each sample a 3–tuple). These streams correspond to two different classes, namely the recording of "looped circular" movements respectively

[1] Perhaps, a 10–fold crossvalidation would have been more adequate, however, a larger multi–fold than a 5–fold was not possible as it gave out–of–memory errors. This is strange, as the amount of required (allocated) memory in principle should be constant with respect to the number of folds (what is needed of additional allocated memory is just a few floating–point numbers for adding up the results per fold – to be averaged in the end), and I suspect this is due to a bug in Weka.

around and along the earth's gravity vector (i.e. horizontal and vertical movements). The two streams were captured/recorded for 59 seconds with a 60 Hz sample–rate, which in total gives 7080 samples (i.e. $7080/2 = 3540$ samples in each stream/class).
In each experiment, instances were generated using a sliding window (segment) of constant length (i.e. constant time–scale). Each new window is shifted/slided only by one sample (time–slot, 3–tuple) from the previous. Window length as measured in number of samples is the only parameter varied in these experiments (constant for each experiment). Moreover, the relation of the window length w to the number of instances $\|\mathcal{D}_w^*\|$ in each class * is simply given by the equation $\|\mathcal{D}_w^*\| = 7080/2 - w + 1 \Leftrightarrow w = 3541 - \|\mathcal{D}_w^*\|$. Regarding notation, here, an instance means a segment—a windowed "snapshot" of a historical part (with constant time–scale) of the stream.

Classification of 167 ms motion segments

Here, a windowsize of ten samples was used (i.e. each instance consisted of $3 \times 10 = 30$ numeric attributes). The dataset consisted of 7060 instances, and all instances were correctly classified. The results are listed in Table 4.1.

Table 4.1: Results from 167 ms motion segments

Class	Precision	Recall
1	100%	100%
2	100%	100%

Classification of 983.3 ms motion segments

Here, a windowsize of 59 samples was used to generate the dataset which here consists of 6962 instances. The results from 5–fold crossvalidation were identical to those of the former experiment, as illustrated in Table 4.1.

Classification of 3 second's motion segments

The dataset for this experiment was generated from the two streams (separately) with a window–size of 180 samples, and therefore consists of 6720 instances. Here, there were only three incorrectly classified instances, hence the accuracy was approximately at 99.96 %. The results are listed in Table 4.2.

Table 4.2: Results from 3 second's motion segments

Class	Precision	Recall
1	99.9%	100%
2	100%	99.9%

Classification of 4167 ms motion segments

This experiment's dataset was generated with a window–size of 250 samples yielding 6580 instances. Here, there were only 61 incorrectly classified instances, yielding an accuracy of 99.07 %. The results are listed in Table 4.3.

Table 4.3: Results from 4167 ms motion segments

Class	Precision	Recall
1	100%	98.1%
2	98.2%	100%

4.2 Experiments with all possible segment lengths on a medium–sized dataset

The following plots in Figures 4.1 and 4.2 are from the same set of experiments with a stream of 300 samples, which correspond to the first range of samples in the same streams as experimented on above. Since evaluation was performed by 10–fold cross validation, all possible segment lengths range from 1 to 289 (can not have more folds than instances).

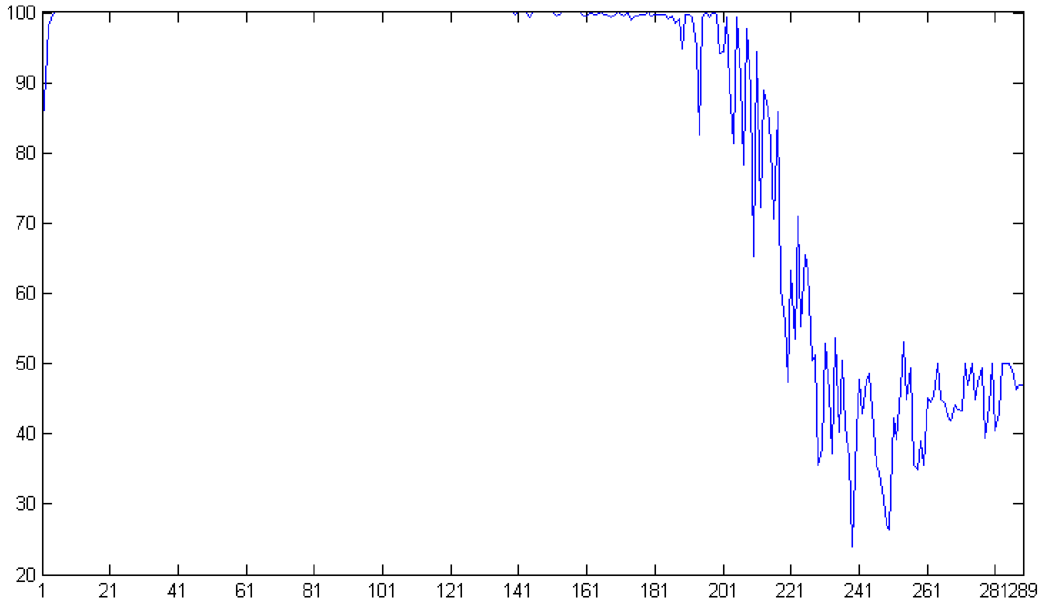

Figure 4.1: This figure shows classification accuracy on the complete range of segment sizes experimented with.

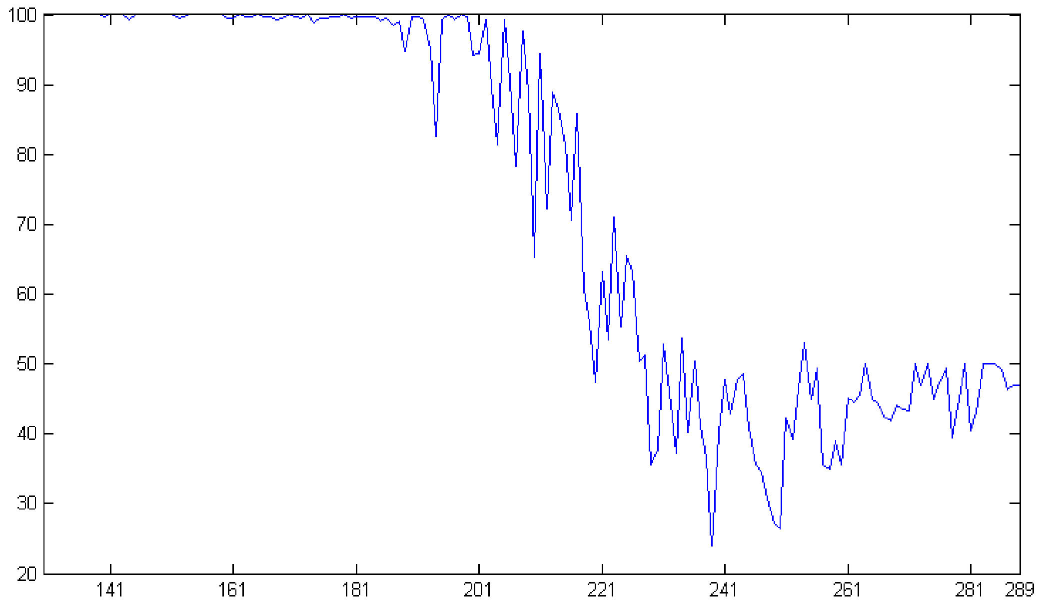

Figure 4.2: This figure shows the more accuracy–varying range of Figure 4.1

Chapter 5

Conclusion

From all the experiments run, accuracy is mostly very near or equal to unity. This hints me that the classification From the four first binary classification experiments evaluated with a 5–fold cross validation, and also the last binary classifying experiments on a significantly larger range of segment lengthsthat gave the most statistically significant results, we have seen that for most segment lengths, the accuracy was 100%.

5.1 Discussion

From the experiments presented, it is fairly obvious to conclude that on most ranges of segment lengths, the classifier was not challenged much by training data derived from the two acceleration streams. Moreover, the average for all the segment lengths was 85.3%. The implicit definition that the same category of fixed–length motion segments exist on all possible fixed–sized substreams on a stream of fixed class, did indeed make classification a trivial task for the classifier. This was unrealistic, especially since only one accelerometer was used for stream capturing. If more accelerometers were used, I assume this would be slightly less unrealistic. This data segmentation method represents an extreme variant in which the machine perception of a motion is tested at an extreme of possible definitions.
In SVM, kernel functions $\mathbf{K}(\mathbf{y}_i, \mathbf{y}_j)$ are used for mapping vectors in input space to vectors in feature space and represent similarity measures. For the performed experiments, the applied kernel function was the RBF function $\mathbf{K}(\mathbf{y}_i, \mathbf{y}_j) = \exp(-\gamma \|\mathbf{y}_i - \mathbf{y}_j\|^2)$. This can be interpreted as the Euclidean distance Chaovalitwongse and Pardalos [2008], which illuminates how it gen-

erally was possible to acchieve such accuracies. Compared to larger segments in each class, one has much more time for moving the accelerometer so that its class–to–class covariance gets much larger than for comparable smaller segments.

Each training set was generated with a sliding window of minimal step–size 1. This generates a maximum number of (overlapping) instances compared to the window size and what is possible of dataset generation. For the first set og experiments, multiplying the number of segments with their segment size and dividing them on the sample rate of 60 Hz gives over a day of data.

It was trained on quite a large, but easily discriminating set of training examples (i.e. the variance on the y–axis is much larger than for the other set of examples), In a later prototype, to better handle more complex datasets and/or to reduce memory use, these features (transformed raw data (series)) can be carried out by a learning machine (this is often necessary to achieve better classification performance), however, research reported in Pylvänäinen [2005] and and my own preliminary results from early experiments with classification of windowed acceleration signals – in *these* cases – (although the data set in my case consists only of a two–category data set of possibly quite easily discriminative examples – i.e. descriminating the variation along the y–axis probably gives a sufficiently generalizing classifier) suggests that this is not necessary (i.e. that three–dimensional dynamic acceleration itself is adequate). However, my own experiments are limited to the classification of basic horizontal and vertical circular movements. Larger experiments (e.g. using a larger amount of gesture classes/categories (and in particular perhaps of a higher complexity)) could of course suggest the opposite (for classifying data from three–dimensional acceleration samples).

Chapter 6

Future works

There is much more that can be done for analyzing sensor data and for active music in general. Adding more classes to the dataset, and generate them from acceleration streams with non–overlapping windows would probably yield more insight.

Moreover, it could be interesting to look at relations between motion and sound with emphasis on the tempo of the music listened to while capturing accelerations seen on multiple time–scales. Also, estimating the direction of gravity has not yet been carried out. Therefore, it may be important to have the accelerometer oriented (and probably to some degree also located) identical (or very similar) as it was during training of the classifier. As a consequence, the accelerometer user may use some time to figure out which orientation the accelerometer should have. A naïve, but non–practical solution for this may be to automatically create rotated versions of each vector used for classifier training, but this would definitely increase the memory use and training time by magnitudes.

Especially, if Ableton supplies Live with a more open API as regards to clip information in the arrangement view (linear composition), further "LiveBot research" would be considerably simplified.

A

Appendix A

SVM Classifier implemented as a Java External for Max

This implementation, whose Java class is named `wml.SvmLM`, is based on the Weka machine learning library and its wrapper for LibSVM, a popular implementation of SVM classification and regression. Upon training the classifier, this Java external performs segmentation on the acceleration sample stream stored in a (user–chosen) *.coll* file for each class. After training, when classifying novel patterns, segmentation is performed outside this external, which in the MaxBot prototype is performed with the interconnected FIFO buffer Java external object named `wml.utils.ListWindow` (i.e. the system user/developer has the opportunity of using other, perhaps faster, segmentation implementations).

```
package wml;


import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectOutputStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.ArrayDeque;
import java.util.Iterator;
import java.util.Random;
import java.util.StringTokenizer;

import com.cycling74.max.*;

import weka.classifiers.Evaluation;
import weka.classifiers.functions.LibSVM; // Optimized, bug-fixed
    version of wlsvm.WLSVM
```

```java
import weka.core.Attribute;
import weka.core.FastVector;
import weka.core.Instance;
import weka.core.Instances;

import org.apache.log4j.Logger;

/*
 * Sources for API and inspiration:
 *      http://weka.sourceforge.net/doc/
 *      http://weka.sourceforge.net/doc/weka/core/Instance.html
 *
 *      http://weka.wikispaces.com/Use+Weka+in+your+Java+code
 *
 *
     http://ianma.wordpress.com/2010/01/16/weka-with-java-eclipse-getting-started/
 *
 *      http://www.cs.iastate.edu/~yasser/wlsvm/
 *
 *      http://shawndra.pbworks.com/f/Weka+filters.pdf
 */

public class SvmLM
    extends MaxObject
{
    // Pragmatics:
    private final boolean maxTest = true; // false ~ JavaTest
    private final boolean DEBUG = true;

    /**
     * Log4j logger
     */
    public static Logger log4j = Logger.getLogger( "wml.SvmLM" );

    // Globals:
    private static final int DEFAULT_DIM_SIZE = 3; // TODO: Remove
        restriction "must be 3n"
    private static final int DEFAULT_WINDOW_SIZE = 10;
    private static final int DEFAULT_CLASS_COUNT = 2;
    private static final int DEFAULT_LABEL = 1;
    private static final int DEFUALT_CAPACITY = 10;

    /* LibSVM options:

          Valid options are:

             -S <int>
             Set type of SVM (default: 0)
             0 = C-SVC
             1 = nu-SVC
             2 = one-class SVM
             3 = epsilon-SVR
             4 = nu-SVR

             -K <int>
             Set type of kernel function (default: 2)
             0 = linear: u'*v
             1 = polynomial: (gamma*u'*v + coef0)^degree
             2 = radial basis function: exp(-gamma*|u-v|^2)
             3 = sigmoid: tanh(gamma*u'*v + coef0)

             -D <int>
             Set degree in kernel function (default: 3)

```

```
            -G <double>
            Set gamma in kernel function (default: 1/k)

            -R <double>
            Set coef0 in kernel function (default: 0)

            -C <double>
            Set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default: 1)

            -N <double>
            Set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default: 0.5)

            -Z
            Turns on normalization of input data (default: off)

            -P <double>
            Set the epsilon in loss function of epsilon-SVR (default: 0.1)

            -M <double>
            Set cache memory size in MB (default: 40)

            -E <double>
            Set tolerance of termination criterion (default: 0.001)

            -H
            Turns the shrinking heuristics off (default: on)

            -W <double>
            Set the parameters C of class i to weight[i]*C, for C-SVC (default: 1)

            -B
            Trains a SVC model instead of a SVR one (default: SVR)

            -D
            If set, classifier is run in debug mode and
            may output additional info to the console
     *
     */
    private static final String[] LIBSVM_CLASSIFIER_OPTIONS =
    {
        "-i", // ?

        //---------------
        "-S",          // LibSVM options:

        "0",           // Classification problem (multi-class SVM a.k.a. C-SVC)
        "-K", "2",     // RBF kernel
        "-G", "1",     // gamma

        "-C", "1",     // C (Complexity Cost), 1 is default (not necessary to set)
        "-B",

        "-Z", "1",     // normalize input data (off by default, here: on)

        "-M", "2000"    // cache size in MB
    };
    private static final String[] VALID_MODES = { "learning",
```

```
        "classifying" };

    private int dimSize;      // Length of feature vector
    private int windowSize;   // Length of window (slots = n *
        dimSize)
    private int classCount;
    private int classIndex;
    private int capacity;
    private int label;
    private int readClassesCount;
    private String[] options;

    private boolean pretrainedClassifier;

    LibSVM svmClassifier;
    Instances trainingSet, testSet;


    // NB: Is called before any attributes are set
    public SvmLM()
        throws Exception
    {
        dimSize = DEFAULT_DIM_SIZE;
        windowSize = DEFAULT_WINDOW_SIZE;
        classCount = DEFAULT_CLASS_COUNT;
        readClassesCount = 0;
        capacity = DEFUALT_CAPACITY;
        label = DEFAULT_LABEL;
        pretrainedClassifier = false;

        declareAttributes( "dimSize", "windowSize", "classCount",
            "capacity", "options", "pretrainedClassifier" );
    }

    public void loadCategoryDataFile( Atom[] fileNamePathMessage )
    {
        String thisCollFilePath = "";

        if ( fileNamePathMessage.length >= 1 )
            for ( int i = 0; i < fileNamePathMessage.length; i++ )
                if ( i == 0 )
                    thisCollFilePath += fileNamePathMessage[ i
                        ].getString();
                else
                    thisCollFilePath += " " + fileNamePathMessage[ i
                        ].getString();

        File collFile = new File( thisCollFilePath );
        FileInputStream fis = null;

        ++readClassesCount;
        try
        {
            fis = new FileInputStream( collFile );
            BufferedReader br = new BufferedReader( new
                InputStreamReader( fis ) );
            // Queue
            boolean windowIsComplete = false;
            int windowSlotsFilled = 0;
            ArrayDeque<Atom> deque = new ArrayDeque<Atom>( dimSize
                * windowSize );

            String lineRead = "";
            int linesRead;
```

```
            for ( linesRead = 0; ( lineRead = br.readLine() ) !=
                null; linesRead++ )
            {
                StringTokenizer tokenizer = new StringTokenizer(
                    lineRead, " ,;" );
                while ( tokenizer.hasMoreTokens() )
                {
                    tokenizer.nextToken(); // "Time tag" (sample
                        index) ignored here
                    for ( int i = 0; i < dimSize; i++ )
                    {
                        String tokenized = tokenizer.nextToken();

                        float val = Float.parseFloat( tokenized );
                        deque.push( Atom.newAtom( val ) );
                    }

                    if ( !windowIsComplete )
                    {
                        if ( ( windowSlotsFilled += dimSize ) ==
                            dimSize * windowSize )
                            windowIsComplete = true;
                    }
                    else
                    {
                        Iterator<Atom> it =
                            deque.descendingIterator();
                        int completeWindowSize = dimSize * windowSize;

                        Atom[] window = new Atom[ completeWindowSize
                            ];
                        for ( int i = 0; i < completeWindowSize; i++ )
                            window[ i ] = it.next();

                        addTrainingInstance( window, ( "" +
                            readClassesCount ) );

                        for ( int d = 0; d < dimSize; d++ )
                            deque.pollLast();
                    }
                }
            }

            properPost
            (
              "Successfully parsed examples from " +
                  thisCollFilePath +
              " and associated them with class index (label) " + (
                  readClassesCount - 1 )
            );

            fis.close();
            br.close();
        }
        catch ( FileNotFoundException e )
        {
            --readClassesCount;
            properExceptionPost( e, "Did not find the file " +
                thisCollFilePath );
        }
        catch ( IOException e )
        {
```

```
            --readClassesCount;
            properExceptionPost( e, "I/O error, i.e. no success
                parsing contents of the file " + thisCollFilePath );
        }
    }

    private void declareAttributes( String ... attNames )
    {
        for ( String attName : attNames )
            declareAttribute( attName );
    }

    public void initClassifier()
        throws Exception
    {
        if ( svmClassifier == null )
          svmClassifier = createLibSvmClassifier();

        doDeclareDataSets
        (
            dimSize,
            windowSize,
            classCount = 1,
            capacity = 2*3540
        );
    }

    private LibSVM createLibSvmClassifier()
    {
        LibSVM classifier = new LibSVM(); // A classifier
            implementing versions of Support Vector Machine

        if ( DEBUG )
          classifier.setDebug( true );

        try
        {
            /* setOptions Javadoc at
             *
                 http://www.java2s.com/Open-Source/Java-Document/Science/weka/weka/class
             * /functions/LibSVM.java.java-doc.htm#setOptionsString
             */
            classifier.setOptions( LIBSVM_CLASSIFIER_OPTIONS );
        }
        catch ( Exception e )
        {
            properExceptionPost( e, "Error setting options for
                LibSVM: " );
        }

        return classifier;
    }

    private void doDeclareDataSets( int dimSize, int windowSize,
        int classCount, int capacity )
    {
        // For each label, declare positive/negative category
            membership
        FastVector classValues = new FastVector( 2 * classCount );
        for ( int label = 1; label <= classCount; label++ )
        {
            classValues.addElement( "" + label );   // Positive
            classValues.addElement( "!" + label );  // Negative
```

```
        }

        int length = ( dimSize * windowSize );

        FastVector wekaAttributes = new FastVector( length + 1 );
        for
        (
            int i = 0, j = 0;
            i < length;
            i += dimSize, j++
        )
        {
            wekaAttributes.addElement
            (
                new Attribute( "X" + j )
            );
            wekaAttributes.addElement
            (
                new Attribute( "Y" + j )
            );
            wekaAttributes.addElement
            (
                new Attribute( "Z" + j )
            );
        }

        wekaAttributes.addElement
        (
            new Attribute( "theClass", classValues )
        );

        // Create empty training set
        trainingSet = new Instances( "3D acceleration training
            set", wekaAttributes, capacity );
        testSet = new Instances( "3D acceleration test set",
            wekaAttributes, capacity );
        trainingSet.setClassIndex( length );
        testSet.setClassIndex( length );
    }

    public void declareDataSets()
    {
      doDeclareDataSets
      (
        dimSize, windowSize, classCount, capacity
      );
    }

    public void trainClassifier()
        throws Exception
    {
        if ( !pretrainedClassifier )
        {
          post( "Training classifier..." );
          doTrainClassifier( svmClassifier, trainingSet );
          post( "Classifier trained." );
          pretrainedClassifier = true;
            savePretrainedClassifier( svmClassifier );
        }
        else
          svmClassifier = loadPretrainedClassifier();
    }

```

```
    public void getSetupForExperiment()
    {
        properPostExperimentalSetup();
    }

    public void evaluateClassifier()
    {
      if ( pretrainedClassifier )
      {
            int numFolds = 5; // Number of folds in
                cross-validation (more folds may cause out-of-memory
                error...)

            Evaluation eval = evaluateCVTrainedClassifier(
                svmClassifier, trainingSet, numFolds );

            properMultiLinePost( eval.toSummaryString(), "Using " +
                numFolds + "-fold cross-validation, we got:" );
            try
            {
        properMultiLinePost( eval.toClassDetailsString(), "Class
            details:" );
      }
            catch ( Exception e )
            {
              if ( DEBUG )
              {
                properExceptionPost( e, "Error calling
                    <Evaluation>.toClassDetailsString(); class is
                    not nominal: " );
              }
      }
      }
    }

    private void doTrainClassifier( LibSVM classifier, Instances
        trainingSet )
    {
        try
        {
            svmClassifier.buildClassifier( trainingSet );
  }
        catch ( Exception e )
        {
            post( "Could not build classifier..." );
            if ( DEBUG )
    e.printStackTrace();
        }
    }

    private Evaluation evaluateCVTrainedClassifier( LibSVM
        classifier, Instances traingSet, int numFolds )
    {
        Random random = new Random( 13 );
        Evaluation eval = null;
        try
        {
            eval = new Evaluation( trainingSet );
            eval.crossValidateModel( svmClassifier, trainingSet,
                numFolds, random );
  }
        catch ( Exception e )
```

```
   {
     if ( DEBUG )
    e.printStackTrace();
}

   return eval;
 }

 private LibSVM loadPretrainedClassifier()
 {
LibSVM pretrainedLibSVM = null;
  post( "Loading pretrained classifier..." );
  try
  {
    pretrainedLibSVM = readPretrainedClassifier();

    post( "Loading completed." );
  }
  catch ( Exception e )
  {
    pretrainedLibSVM = new LibSVM();
    post( "Could not load pretrained classifier. Reverted to non-trained classifier (and set pretrainedClassifier to 'false')." );

    pretrainedClassifier = false;
    if ( DEBUG )
    e.printStackTrace();
  }

  return pretrainedLibSVM;
 }

 private void savePretrainedClassifier( LibSVM svmClassifier )
 {
   try
   {
     ObjectOutputStream oos =
       new ObjectOutputStream
       (
         new FileOutputStream(
           "lastSavedClassifierModel.dat" )
       );

     oos.writeObject( svmClassifier );
     oos.flush();
     oos.close();
   }
   catch ( FileNotFoundException e )
 {
      post( "File not found." );

      if ( DEBUG )
        e.printStackTrace();
 }
 catch ( IOException e )
 {
   post( "I/O error. Perhaps, there is no more disk space?" );

   if ( DEBUG )
     e.printStackTrace();
 }
```

```
    }

    private LibSVM readPretrainedClassifier()
        throws Exception
    {
        return (LibSVM) weka.core.SerializationHelper.read( "lastSavedClassifierModel.dat" );
    }

    private void properPostExperimentalSetup()
    {
      properPost
    (
        "Setup for this experiment:"
        );
        properPost
        (
            "\t" + "The training set is based on a " + ( (int) Math.pow( 2 , classCount ) ) + "-category dataset/stream of " +
            dimSize + "-dimensional instances."
    );
        properPost
        (
            "\tThe actual training set (in feature space) consists of the same data \"time-windowed\"/augmented " +
            "into (\"chunked\") vectors of correspondingly larger dimensionality "
    );
        properPost
        (
            "\t(here, " + windowSize + " samples (of " + dimSize + "-dimensional instance-vectors) in each augmented vector)."
    );
    }

    private void properPost( String message )
    {
        if ( maxTest )
            post( message );
        else // javaTest (e.g. JUnit testing)
            log4j.debug( message );
    }

    /**
     * Callback method for the parent mxj object for receiving lists
     */
    public void list( Atom[] vec )
    {
        if ( pretrainedClassifier )
            classifyInstance( vec );
        else
            addTrainingInstance( vec, ( "" + label ) );// XXX FixMe
    }

    /**
     * Method for adding an instance to the trainingSet
     *
     * @param vec Max list assumed to be a real vector
     */
    private void addTrainingInstance( Atom[] vec, String label )
```

```
    {
        int completeWindowSize = dimSize * windowSize;

        Instance instance = new Instance( completeWindowSize + 1 );
            // one for the label as well
        instance.setDataset( trainingSet );

        for ( int attIndex = 0; attIndex < completeWindowSize;
            attIndex++ )
        {
            float value = vec[ attIndex ].getFloat();
            instance.setValue( attIndex, value );
        }

        // XXX FixIt
        if ( label.equals( "2" ) )
            label = "!1";

        instance.setValue( completeWindowSize, label );

        trainingSet.add( instance );
    }

    private void classifyInstance( Atom[] vec )
    {
        Instance testInstance = new Instance( vec.length );
        testInstance.setDataset( testSet );

        for ( int attIndex = 0; attIndex < vec.length; attIndex++ )
            testInstance.setValue( attIndex, vec[ attIndex
                ].getFloat() );

        double predictedClassIndex = -1.0;
        try
        {
            predictedClassIndex = svmClassifier.classifyInstance(
                testInstance );
        }
        catch ( Exception e )
        {
            String message = "An error occured upon classification.
                Output (erroneous) class index -1";

            if ( DEBUG )
    properExceptionPost( e , message );
            else
    post( message );
        }

        outputPredictedClassIndex( 0 , predictedClassIndex );
    }


    private void reStart()
    {
        // TODO Implement reStart() ?
    }

    /**
     * Max setter method:
     * Usage: message/@argument classCount <int>
     * @param  newClassCount
     */
```

```
    public void classCount( Atom[] newClassCount )
    {
        Atom arg;
        if ( newClassCount.length >= 1 )
        {
            arg = newClassCount[ 0 ];

            if ( arg.isInt() )
                doSetClassCount( arg.getInt() );
            else
                properPost
                (
                    "Error in setClasscount <classCount> message: " +
                    "<classCount> must be a natural number."
                );
        }
    }

    /**
     * Max setter method:
     * Usage: message/@argument pretrainedClassifier <boolean>
     * @param  usePretrainedClassifier
     */
    public void pretrainedClassifier( Atom[] usePretrainedClassifier )
    {
      post( ".........." ); // TODO (DEBUG) Remove this line
        Atom arg;
        if ( usePretrainedClassifier.length >= 1 )
        {
            arg = usePretrainedClassifier[ 0 ];
            String message = arg.getString();

            if ( message.equalsIgnoreCase( "true" ) )
            {
                try
                {
                    pretrainedClassifier = true;
                    trainClassifier(); // Loads pretrained classifier (does not really train it again)
                    initClassifier();

                    post( "Using pretrained classifier" );

        }
                catch ( Exception e )
                {
                  pretrainedClassifier = false;
                  post( "Could not use pretrained classifier (pretrainedClassifier set to false)" );

                  if ( DEBUG )
                    e.printStackTrace();
        }
            }
            else if ( message.equalsIgnoreCase( "false" ) )
                pretrainedClassifier = false;
            else
                post( "Error: The parameter after 'pretrainedClassifier' must be a boolean, true or false." );
        }
    }
```

```java

    /**
     * Max getter method - call--result output from Max info outlet:
     */
    public void pretrainedClassifier()
    {
        output( getInfoIdx(), pretrainedClassifier );
    }

    private void doSetClassCount( int newClassCount )
    {
        if ( classCount != newClassCount )
            classCount = newClassCount;
    }

    public void getClassCount()
    {
        output( getInfoIdx(), classCount );
    }

    @Deprecated
    /** Not necessary.
     * [classIndex #] does not call this method.
     */
    public void setClassIndex( Atom[] nextClassIndex )
    {
        Atom arg;
        if ( nextClassIndex.length >= 1 )
        {
            arg = nextClassIndex[ 0 ];

            if ( arg.isInt() )
                doSetClassIndex( arg.getInt() );
            else if ( arg.isFloat() )
                doSetClassIndex( (int) arg.getFloat() );
            else
                properPost
                (
                    "Error in handling setClassIndex <classIndex> message: " +
                    "<classIndex> must be a positive integer."
                );
        }
    }

    @Deprecated
    private void doSetClassIndex( int nextClassIndex )
    {
        if ( nextClassIndex != classIndex )
        {
            classIndex = nextClassIndex;

            reStart();
        }
    }

    public void getClassIndex()
    {
        output( getInfoIdx(), classIndex );
    }

    @Deprecated
    /** Not necessary.
```

```java
     * [dimSize #] does not call this method.
     */
    public void setDimSize( Atom[] newDimSize )
    {
        Atom arg;
        if ( newDimSize.length >= 1 )
        {
            arg = newDimSize[ 0 ];

            if ( arg.isInt() )
                doSetDimSize( arg.getInt() );
            else if ( arg.isFloat() )
                doSetDimSize( (int) arg.getFloat() );
            else
                properPost
                (
                    "Error in handling setDimSize <dimSize> message: " +
                    "<dimSize> must be a natural number."
                );
        }
    }

    @Deprecated
    private void doSetDimSize( int newDimSize )
    {
        if ( newDimSize > 0 && newDimSize != dimSize )
        {
            dimSize = newDimSize;

            reStart();
        }
    }

    public void getDimSize()
    {
        output( getInfoIdx(), dimSize );
    }

    @Deprecated
    /** Not necessary.
     * [capacity #] does not call this method,
     * only [setCapacity #] does.
     */
    public void setCapacity( Atom[] nextCapacity )
    {
        if ( nextCapacity.length > 0 )
        {
            Atom arg = nextCapacity[ 0 ];

            if ( arg.isInt() )
                doSetCapacity( arg.getInt() );
            else if ( arg.isFloat() )
                doSetCapacity( (int) arg.getFloat() );
            else
                properPost
                (
                    "Error in handling setCapacity <capacity> message: " +
                    "<capacity> must be a natural number."
                );
        }
    }
```

```
    @Deprecated
    private void doSetCapacity( int newCapacity )
    {
        capacity = newCapacity;
    }

    public void getCapacity()
    {
        output( getInfoIdx(), capacity );
    }

    @Deprecated
    /** Not necessary.
     * [windowSize #] does not call this method,
     * only [setWindowSize #] does. */
    public void setWindowSize( Atom[] nextWindowSize )
    {
        if ( nextWindowSize.length >= 1 )
        {
            Atom arg = nextWindowSize[ 0 ];

            if ( arg.isInt() )
                doSetWindowSize( arg.getInt() );
            else if ( arg.isFloat() )
                doSetWindowSize( (int) arg.getFloat() );
            else
                properPost
                (
                    "Error in handling setWindowSize <windowSize>
                        message: " +
                    "<windowSize> must be a natural number."
                );
        }
    }

    @Deprecated
    private void doSetWindowSize( int newWindowSize )
    {
        windowSize = newWindowSize;
    }

    public void getWindowSize()
    {
        output( getInfoIdx(), windowSize );
    }

    // TODO Test setOptions( Atom[] newOptions ). Should be
        deprecated (only use [options %s]?)
    public void options( Atom[] newOptions )
    {
        if ( newOptions.length >= 1 )
        {
            String[] oldOptions = options.clone();

            options = new String[ newOptions.length ];
            for ( int i = 0; i < newOptions.length; i++ )
            {
                options[ i ] = newOptions[ i ].getString();
            }

            try
            {
```

```
                svmClassifier.setOptions( options );
            }
            catch ( Exception e )
            {
                properPost( "Error setting classifier options: " +
                    e.getStackTrace().toString() );

                // Exception handling (revert to old options)
                options = oldOptions.clone();
            }
        }
    }

    public void options()
    {
    output
    (
        getInfoIdx() ,
        ( ( options != null && options[0] != null ) ?
            options.toString() : "Not set" )
    );
    }

    private void output( int outletIndex, String message )
    {
        if ( maxTest )
            outlet( outletIndex, message );
        else
            log4j.debug( message );
    }

    private void output( int outletIndex, int integer )
    {
        if ( maxTest )
            outlet( outletIndex, integer );
        else
            log4j.debug( integer );
    }

    private void output( int outletIndex, boolean bool )
    {
        if ( maxTest )
            outlet( outletIndex, bool );
        else
            log4j.debug( bool );
    }

    private void outputPredictedClassIndex( int outletIndex, double
        predictedClassIndex )
    {
      int message = (int) predictedClassIndex;

      if ( maxTest )
        outlet( outletIndex, message );
      else
        log4j.debug( "" + message );
    }

    private void properStringArrayPost( String header, String[]
        stringArray )
    {
        properPost( header );
```

```

        for ( int i = 0; i < stringArray.length; i++ )
            properPost( stringArray[ i ] );
    }

    private void properExceptionPost( Exception e, String header )
    {
        String[] stackTraceLines = stackTraceToString( e ).split(
            "\\n" );
        if ( header != null)
            properPost( header );
        for ( int i = 0; i < stackTraceLines.length; i++ )
            properPost( stackTraceLines[ i ] );
    }

    private void properMultiLinePost( String content, String header
        )
    {
        if ( header != null)
            properPost( header );

        String[] stackTraceLines = content.split( "\\n" );
        if ( stackTraceLines != null )
            for ( int i = 0; i < stackTraceLines.length; i++ )
                properPost( stackTraceLines[ i ] );
    }

    private String stackTraceToString( Exception e )
    {
        // Source: http://www.rgagnon.com/javadetails/java-0029.html
        try
        {
            StringWriter sw = new StringWriter();
            PrintWriter pw = new PrintWriter( sw );

            e.printStackTrace( pw );

            return "------\r\n" + sw.toString() + "------\r\n";
        }
        catch ( Exception bad )
        {
            return "Bad printStack";
        }
    }

}
```

Appendix B

JavaScript External for auto–triggering Live Clips

For an overview of the Live API web [2010b], see the Live Object Model illustrated in Figure B.

The JavaScript implementation of LiveBot utilizes the LiveAPI JavaScript object web [2010a] as follows.

```
/**
 *  @projectDescription
 *   This is an active music approach for Ableton Live using the
    LiveAPI for JS in Max/MSP (Max for Live).
 *   The script reads each tracks' clip names that control much of
    the playback that starts
 *   when receiving current beat position on the inlet of a Max JS
    external.
 *
 *  @author Roger S. Grading
 *  @version 0.5a Build 1
 */


// I/O
inlets = 1;
outlets = 2;


// Debug on/off:
/** Decides whether to post debug messages to the (Max) console */
var DEBUG = true;


/** Decides whether to process only the first track
 *  (JavaSscript code execution in Max is slow!) or all of them
 */
var MINITEST = true;
var TEST_TRACK_INDEX = 0;



```

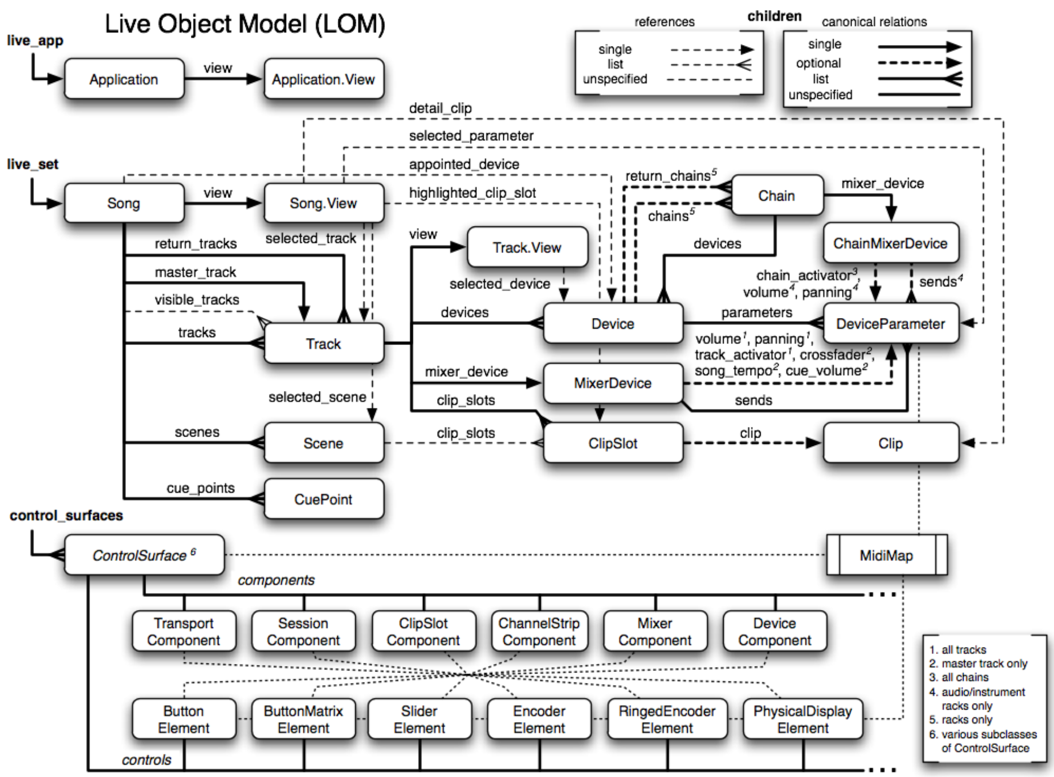

Figure B.1: The (Ableton) Live Object Model (API overview). Image taken from web [2010c].

```
// Global variables:
/** Root "live_set" LiveAPI object
 *
 *  @type {LiveAPI} song
 */
var song = new LiveAPI( this.patcher, "live_set" );


/** The number of tracks in the Live set
 *
 *  @type {Number} n_tracks
 */
var n_tracks;


/** 2D array of Clips
 *
 *  @type {Array<Array<Clip>>} tracks_Clips
 */
var tracks_Clips;


/** 2D array of clip indexes
 *
```

```
 *    (where tracks_clips_ix[ track_index ][ count-1 ] = Clip Slot
     index)
 *
 *  @type {Array<Array<Number>>} tracks_clips_ix
 */
var tracks_clips_ix;


/** 2D array of clip names
 *
 *    (where tracks_clip_names[ track_index ][ count-1 ] = Clip
     name)
 *
 *  @type {Array<Array<String>>} tracks_clips_names
 */
var tracks_clips_names;


/** 2D array of eah track's playing/active Clip Slot index
 *
 *  @type {Array<Array<Number>>} playing_tracksClip_ix
 */
var playing_tracksClip_ix;


/** The beats left to play each track's active clip
 *  (i.e. the "beat-times" before each track's next clip is played)
 *
 *  @type {Number} local_clips_beatCounters
 */
var local_clips_beatCounters;


/** Current Live (song) beat
 *
 *  @type {Number} beats
 */
var beat = 1; // Assuming the first beat of the song

var clips;
var banged = false;


// Debug settings init:
if ( DEBUG )
  initDebugSettings();


/** Function call thread priority
 *    1 - High
 *    0 - Low (default)
 */
bang.immediate = 1;


/** Gets called when a bang is received in the inlet of the "js"
    Max external */
function bang()
{
  processTracks();

  banged = true;
}

```

```

/** Gets called when an int is received in the inlet of the "js"
    Max external
 *
 * @param {Number} beat  The beat position of the Live set (song)
 */
function msg_int( beat )
{
    this.beat = beat;
  post( "Beat#: " + this.beat + "\n" );

  if ( banged )
  {
    updateClipManager( beat );
  }
  else
  {
    post( "** LiveController: Has no effect until bang is received
        at my inlet **\n" );
  }
}


/** Reads clips from each track's clip slots */
function processTracks()
{
  /** Get array with all track id's
   *
   *    Format: (id <track_id_1> ... id <track_id_n>)
   *
   *  @private
   *  @type {Array<String>} tracks_IDs
   */
  var tracks_IDs = song.get( "tracks" );

  n_tracks = song.getcount( "tracks" );

  playing_tracksClip_ix    = new Array( n_tracks ); // Each tracks'
      clip progression reflected by active index
  tracks_clips_ix      = new Array( n_tracks );
  tracks_clips_names    = new Array( n_tracks );
  local_clips_beatCounters  = new Array( n_tracks );
  tracks_Clips      = new Array( n_tracks );

  for ( var track_ix = ( MINITEST ? TEST_TRACK_INDEX : 0 );
      track_ix < ( MINITEST ? ( TEST_TRACK_INDEX + 1 ) : n_tracks
      ); track_ix++ )
  {
    if ( DEBUG )
    {
      //post( "ClipNames for track #" + track_ix + " :: " );
    }

    tracks_clips_ix    [ track_ix ] = new Array(); // Size yet
        unknown
    tracks_clips_names   [ track_ix ] = new Array();
    tracks_Clips     [ track_ix ] = new Array();

    playing_tracksClip_ix [ track_ix ] = 0; // Don't know that, but
        assume so..
    local_clips_beatCounters[ track_ix ] = 0;

    var track = new LiveAPI( this.patcher, "live_set tracks " +
```

```
        track_ix );
    //var clipSlots = track.get( "clip_slots" ); // no use for this yet
    var n_clipSlots = track.getcount( "clip_slots" );
    var n_clips = processClipSlots( track_ix, n_clipSlots );

    if ( DEBUG )
    {
      post( "\n\nCalling dispResult( track_ix = " + track_ix + ", n_clips = " + n_clips + ")\n\n" );
      dispResults( track_ix, n_clips );
    }
  }
}


function dispResults( track_ix, n_clips )
{
  for ( var i = 0; i < n_clips; i++ )
  {
    clipSlot_ix = tracks_clips_ix[ track_ix ][ i ];

    var myClip = tracks_Clips[ track_ix ][ clipSlot_ix ];
    if (myClip != null)
      post( "Clip[ " + track_ix + " ][ " + i + " ].isDummy() == " + ( myClip.isDummy() ? 1 : 0 ) + "\n" );
  }
}


/** Iterates a track's clip slots
 *
 * @param {Number} track_ix  The index of the track
 * @param {Number} n_clipSlots The number of clip slots in the track
 */
function processClipSlots( track_ix, n_clipSlots )
{
  var k_clipNamesTagged = 0;

  for ( var clipSlot_ix = 0; clipSlot_ix < n_clipSlots; clipSlot_ix++ )
  {
    var clipSlot = new LiveAPI
    (
      this.patcher,
      "live_set tracks " + track_ix + " clip_slots " + clipSlot_ix
    ); // LiveAPI

    var clipID = ( clipSlot.get( "clip" ) )[ 1 ];

    if (clipID != 0) // Clip lives in clipSlot
    {
      var clipNameIsTagged =
      processClip
      (
        track_ix,
        clipSlot_ix,
        k_clipNamesTagged
      );

      if ( clipNameIsTagged )
      {
```

```
        tracks_clips_ix[ track_ix ][ k_clipNamesTagged++ ] =
            clipSlot_ix;
        post( "** tracks_clips_ix[ track_ix == " + track_ix + " ][
            k_clipNamesTagged++ == " + (k_clipNamesTagged-1) + "++
            ] == clipSlot_ix == " + clipSlot_ix + "**\n " );

        if ( k_clipNamesTagged == 1 )
        {
          playing_tracksClip_ix[ track_ix ] = ( k_clipNamesTagged -
              1 ); // ? -1?
        }
      }
    }
    else
    {
      if (DEBUG)
      {
      //  post( "T" + track_ix + ":S" + clipSlot_ix + ":C" + clipID
          + "\n" );
      //post( "* " );
      }
    }
  }
  tracks_clips_ix[ track_ix ][ k_clipNamesTagged ] = -1; //
      Inserting end-tale (-1 an invalid/dummy index)

  if ( DEBUG )
  {
    //post( "\n" );
    dispRelevantClipSlots( track_ix, k_clipNamesTagged );
  }

  return k_clipNamesTagged;
}


function dispRelevantClipSlots( track_ix, k_clipNamesTagged )
{
  for ( var i = 0; i < k_clipNamesTagged; i++ )
    post
    (
      "tracks_clips_names[ " + track_ix + " ][ " + i + " ] = " +
          tracks_clips_names[ track_ix ][ i ] +
        " / " +
      "tracks_clips_ix[ " + track_ix + " ][ " + i + " ] = " +
          tracks_clips_ix[ track_ix ][ i ] +
        "\n"
    );
}


/** Processes a track's clip slot's Clip
 *
 * @param  {Number}  track_ix      The index of the track
 * @param  {Number}  clipSlot_ix    The index of the clip slot
 * @param  {Number}  k_clipNamesTagged  The what?
 * @return  {Boolean} clipNameIsTagged  Decides whether the
     corresponding clip name is tagged
 */
function processClip( track_ix, clipSlot_ix, k_clipNamesTagged )
{
  var clipNameIsTagged = false;
```

```

  var clipObj = new LiveAPI( this.patcher, "live_set tracks " +
      track_ix + " clip_slots " + clipSlot_ix + " clip" );
  var clipName = clipObj.getstring( "name" );

  if ( clipName ) // clipName is defined
  {
    clipNameIsTagged = isTagged( clipName );
    if ( clipNameIsTagged )
    {
      tracks_clips_ix[ track_ix ][ k_clipNamesTagged ] =
          clipSlot_ix;

      parseClipNameTags( track_ix, clipSlot_ix, k_clipNamesTagged,
          clipName );
      tracks_clips_names[ track_ix ][ k_clipNamesTagged ] =
          clipName;
    }
  }

  return clipNameIsTagged;
}


/** Parses tags of a clip name (assuming clip name is tagged)
 *
 *    Valid track types:
 *
 *      K - kickbass (drum)
 *      B - bass
 *      DK - drum kit
 *      M - melody
 *      SFX - sound effect
 *
 * @param {Number} track_ix      The index of the track
 * @param {Number} clipSlot_ix    The index of the clip slot
 * @param {Number} k_clipNamesTagged  The index to use as second
     index in the clip data arrays
 * @param {String} clipName      The name of the clip
 */
function parseClipNameTags( track_ix, clipSlot_ix,
    k_clipNamesTagged, clipName )
{
  // DUMMY_pausebeats
  // _a_beats (where length = beats)
  // _a_length_beats
  // _a_b_length_beats
  var trackType = "";
  var beats = 0;
  var length = -1; // assuming dummy Clip

  var split = clipName.split( "_" );
  post( "\n\"" + clipName + "\".split( \"_\" ) = " + split + " : "
      );

  var intFreq = 0;
  for ( var i = 0; i < split.length; i++ )
  {
    var result = parseInt( split[ i ], 10 );
    if ( isNaN( result ) == false ) // split[ i ] has a valid
        number tag
    {
      intFreq++;
```

```
      post( result + "," );
    }
  }
  post( "\n" );

  if ( intFreq == 2 )
  {
    length = split[ ( split.length - 2 ) ];
    post( clipName + "-CASE2-length: " + length + "\n" );
  }

  beats = split[ ( split.length - 1 ) ]; // assuming intFreq > 0
      (i.e. no syntax errors in clip names)

  if ( intFreq == 1 && split[ 0 ].toUpperCase() != "DUMMY" )
    length = beats;

  tracks_Clips[ track_ix ][ k_clipNamesTagged ] = new Clip(
      length, beats );

  post( clipName + "-DEFAULT-beats: " + beats + "\n" );

  post( "\n" );
}


/** Checks whether clipName is (correctly) tagged
 *
 * @param {String} clipName  The name of the clip
 */
function isTagged( clipName )
{
  // ClipName has tags
  var clipNameIsTagged = false;

  if ( clipName.length > 0 && ( ( clipName.charAt( 0 ) == '_' ) ||
      ( clipName.length > 4 &&
      (clipName.substring(0,5)).toLowerCase() == "dummy" ) ) )
    clipNameIsTagged = true;

  return clipNameIsTagged;
}


/** Updates the state for the clip (playback) manager
 *
 * @param {Number} beat  The song's beat position
 */
function updateClipManager( beatPosition )
{
  for
  (
    var track_ix = ( MINITEST ? TEST_TRACK_INDEX : 0 );
    track_ix < ( MINITEST ? ( TEST_TRACK_INDEX + 1 ) : n_tracks );
    track_ix++
  )
  {
    var clipSlot_LUT_ix = playing_tracksClip_ix[ track_ix ];
    /*
    var init_ClipSlot = new LiveAPI
    (
      this.patcher,
      "live_set tracks " + track_ix + " clip_slots " +
```

```
        tracks_clips_ix[ track_ix ][ clipSlot_LUT_ix ]
    );
    init_ClipSlot.call( "fire"); // Fire clip at initial clip slot

      */
    var clipObj = tracks_Clips[ track_ix ][ clipSlot_LUT_ix ];
    var ongoingBeatsLeft = clipObj.ongoingBeatsLeft--;
    post( ongoingBeatsLeft + " == ongoingBeatsLeft @
        clipSlot_LUT_ix == " + clipSlot_LUT_ix + "\n" );

    if ( ongoingBeatsLeft == 0 || ( clipObj.length == 1 &&
        ongoingBeatsLeft < 1 ) )
    {
      // Advance, update next clip (ix) and fire it
      var next_LUT_ix = ++playing_tracksClip_ix[ track_ix ];

      var next_ClipSlot = new LiveAPI
      (
        this.patcher,
        "live_set tracks " + track_ix + " clip_slots " +
            tracks_clips_ix[ track_ix ][ next_LUT_ix ]
      );

      if ( !next_ClipSlot )
      {
        post( "** LiveController : Could not access ClipSlot object
            @ [ live_set tracks " + track_ix + " clip_slots " +
            tracks_clips_ix[ track_ix ][ next_LUT_ix ] + " ]! **\n"
            );
      }
      else
      {
        next_ClipSlot.call( "fire" );
        post( "** LiveController : next clip fired **\n" );
      }
    }
    else
    {
      if ( clipSlot_LUT_ix == 0 && beat == 0 ) // beat == 0 in
          itself is probably sufficient..
      {
        var next_ClipSlot = new LiveAPI
        (
          this.patcher,
          "live_set tracks " + track_ix + " clip_slots " +
              tracks_clips_ix[ track_ix ][ playing_tracksClip_ix[
              track_ix ] ]
        );
        next_ClipSlot.call( "fire" );
      }
    }
  }
}


/** Initializes debug configuration */
function initDebugSettings()
{
  autowatch = 1;

  post( "** LiveController: Compiled and loaded **\n" );
  bang();
```

```
  post( "** LiveController: returned from bang() call **\n" );
}


Clip.immediate = 1;
/** @constructor Creates a Clip object
 *
 * @param {Number} length The Clip's original length
 * @param {Number} beats The Clip's duration in beats
 */
function Clip( length, beats )
{
  this.length = length;
  this.beats = beats;
  this.ongoingBeatsLeft = beats;
  this.isDummy = function () { return ( length == -1 ); };
}
```

Bibliography

Ableton Live. http://www.ableton.com/live, a. 22

LIBSVM – A Library for Support Vector Machines. http://www.csie.ntu.edu.tw/~cjlin/libsvm/, b. 7

Max for Live. http://www.ableton.com/maxforlive, c. 22

Cycling '74. http://cycling74.com/, d. 3

Phidgets Inc. – Unique and Easy to Use USB Interfaces. http://www.phidgets.com/, e. 4

SensorWiki.org. http://www.sensorwiki.org/, f. 11

Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/, g. 7, 30

weka – LibSVM. http://weka.wikispaces.com/LibSVM, h. 7, 30

Digital audio workstation. http://en.wikipedia.org/wiki/Digital_audio_workstation. 22

Analog Devices ADXL330 accelerometer datasheet. http://www.analog.com/static/imported-files/data_sheets/ADXL330.pdf, 2007. 4, 11

Live API Object. http://www.cycling74.com/docs/max5/vignettes/js/jsliveapi.html, 2010a. 59

Live API Overview. http://www.cycling74.com/docs/max5/refpages/m4l-ref/m4l_live_api_overview.html, 2010b. 59

LOM – The Live Object Model. http://www.cycling74.com/docs/max5/refpages/m4l-ref/m4l_live_object_model.html, 2010c. viii, 60

Sensing Music-related Actions (2008-2012). http://www.fourms.uio.no/projects/sma/index.html, 2010d. 1

Mathematical notation for elementwise multiplication discussed on Physicsforum.com. http://www.physicsforums.com/showthread.php?t=440675, 2010e. 30

Web links for Active Music applications. http://fourms.wiki.ifi.uio.no/Active_Music, 2010f. 9

Support vector machine. http://en.wikipedia.org/wiki/Support_vector_machine, 2010a. 15, 30

Envelope following. http://en.wikipedia.org/wiki/Envelope_detector#Audio, 2010b. viii, 31, 32

Artificial neural network. http://en.wikipedia.org/wiki/Artificial_neural_network, 2011a. vii, 14

Dynamic time warping. http://en.wikipedia.org/wiki/Dynamic_time_warping, 2011b. 5

Hidden Markov model. http://en.wikipedia.org/wiki/Hidden_Markov_model, 2011c. 5

Kinect. http://en.wikipedia.org/wiki/Kinect, 2011d. 11

Pattern recognition. http://en.wikipedia.org/wiki/Pattern_recognition, 2011e. 7

Stereoscopy. http://en.wikipedia.org/wiki/Stereoscopy, 2011f. 11

W. Chaovalitwongse and P. Pardalos. On the time series support vector machine using dynamic time warping kernel for brain activity classification. *Cybernetics and Systems Analysis*, 44:125–138, 2008. ISSN 1060-0396. URL http://dx.doi.org/10.1007/s10559-008-0012-y. 10.1007/s10559-008-0012-y. 6, 37

Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000. ISBN 0471056693. 15, 18

A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, October 2008. ISBN 3540401849. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540401849`. 12

Christophe Gisler. Symbios art – a reactive painting based on voice input and image selection. Master's thesis, University of Fribourg, Switzerland, March 2008. 16

Jeff Hawkins and George Dileep. *Hierarchical Temporal Memory: Concepts, Theory, and Terminology*. `http://www.numenta.com/Numenta_HTM_Concepts.pdf`, 2007. 13

Mats Høvin, Marianne Garder, Rolf Inge Godøy, Jim Tørresen, and Aleksander Refsum Jensenius. *Sensing Music-related Actions*, 2007. 3, 10

Alexander Refsum Jensenius. *Action–Sound : Developing Methods and Tools for Studying Music-Related Bodily Movement*. PhD thesis, Department of Musicology, University of Oslo, 2007. 2

Tod Machover. *Shaping Minds Musically*. *BT Technology Journal*, 22(4): 171–179, 2004. ISSN 1358-3948. doi: http://dx.doi.org/10.1023/B:BTTJ.0000047596.75297.ee. i, 10

Tom M. Mitchell. *Machine Learning (Mcgraw-Hill International Edit)*. McGraw-Hill Education (ISE Editions), 1st edition, October 1997. ISBN 0071154671. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0071154671`. 1, 3

Kristian Nymoen. *Motion tracking in musical instrument interfaces: A discussion of methods for measuring and registering gesture data in musical performances*. Semester assignment, MUS4687 - Special Syllabus in Musicological Modules 3, 2007. 11

Timo Pylvänäinen. *Accelerometer Based Gesture Recognition Using Continuous HMMs*. Pylvänäinen, Timo, 2005. doi: 10.1007/11492429_77. URL `http://dx.doi.org/10.1007/11492429_77`. 5, 7, 38